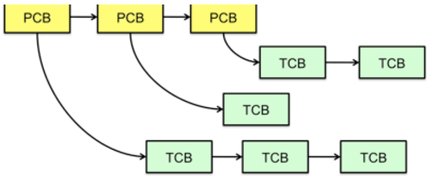
L1 Overview

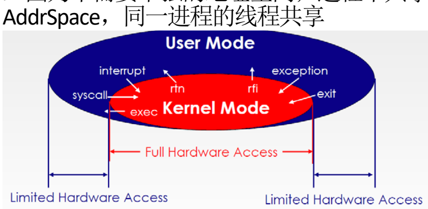
① What's an OS? •Special layer of SW
− Provides application SW access to HW resources
− Convenient abstraction of complex HW devices
− Protected access to shared resources
− Security and authentication
− Communication amongst logical entities

L2 Kernel Process

① Three roles of an OS *1.1* *P22* Referee, Illusionist, Glue.
② Four Basic OS Concepts (process, address space, thread, dual-mode operation)
③ TCB (Thread Control Block) •Holds contents of registers when thread is not running
(For PCB, *vol2 4.7 P52*)
− user threads



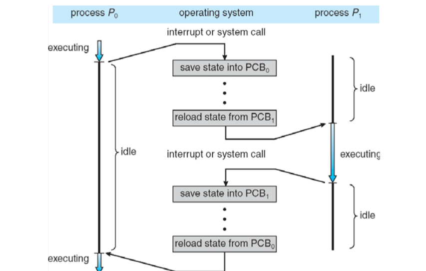
④ Process: Execution environment with Restricted Rights (*Application instance consists of one or more processes)
− (Protected) Address Space with One or More Threads
− Owns memory (address space)
− Owns file descriptors, file system context, ...
− Encapsulate one or more threads sharing processes resources
Thread: a sequential execution stream within process ("Lightweight process"), unit of concurrent execution.
⑤ Threads encapsulate concurrency: "Active" component; Address spaces encapsulate protection: "Passive" part 进程不易建立因为需要复制父进程的内存分配，线程易于建立因为不需要单独的地址空间；进程不共享AddrSpace，同一进程的线程共享

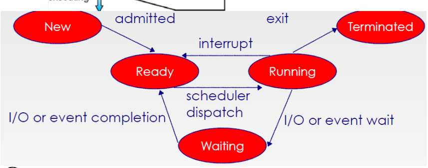


⑥ Virtualized memory is only associated with processes but not with threads. But a distinct virtualized processor is associated with each and every thread. / Processes have overheads(开销) but not threads.

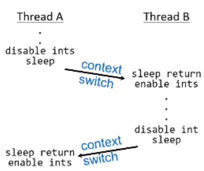
L3 Processes, Fork, System Calls

① Context Switch

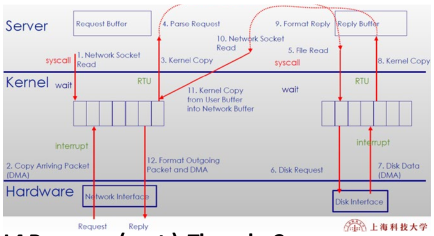




② Superscalar processors can execute multiple instructions that are independent.
Hyper threading duplicates register state to make a second "thread", allowing more instructions to run.



③ Web Sever



L4 Processes(cont.), Threads, Concurrency

① POSIX: Portable Operating System Interfaces in UNIX. Interface for application programmers.
② Thread State
State shared by all threads in process/address space
− Content of memory (global variable, heap)
− I/O state (file descriptors, network connection, etc.)
State "private" to each thread
− Kept in TCB (thread control block)
− CPU registers (including PC)
− Execution stack
Execution Stack
− Parameters, temporary variables
− Return PCs are kept while called procedures are executing
③ Thread context switch cheaper: No need to change address space.
④

一次一个	Process	Thread	Multicore
Switch overhead	High	Medium	Low
Cpu state	Low	Low	Only
Memory/io state	High		
Process creation	High	Medium	Low
Protection: cpu	Yes	Yes	Yes
Protection:mem/io	Yes	No	No
Sharing overhead	High	Low	Low

⑤ An interrupt is a HW-invoked context switch.
⑥ Cooperating Threads Advantage: Share resources, Speedup, Modularity.
⑦ Multithreading Models: Many-to-One, One-to-One, Many-to-Many. (Two-Level Model)

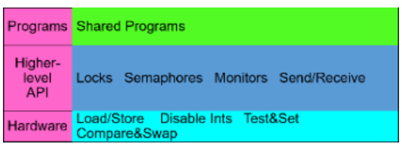
L5 Concurrency and Mutual Exclusion

① Synchronization: using atomic operations to ensure cooperation between threads.
- Mutual Exclusion: ensuring that only one thread does a particular thing at a time.
- Critical Section: piece of code that only one thread can execute at once. Only one thread at a time will get into this section of code.
② Dispatcher(调度员) gets control in two ways
- Internal: Thread does something to relinquish the CPU
- External: Interrupt cause dispatcher to take CPU
③ Lock

```
int value = FREE;

Acquire() {
   disable interrupts;
   if (value == BUSY) {
      put thread on wait queue;
      Go to sleep();
      // Enable interrupts?
   } else {
      value = BUSY;
   }
   enable interrupts;
}

Release() {
   disable interrupts;
   if (anyone on wait queue) {
      take thread off wait queue
      Place on ready queue;
   } else {
      value = FREE;
   }
   enable interrupts;
}
```



在 acquire 中间开启中断会发生什么
• Before Putting thread on the wait queue?
• Release can check the queue and not wake up thread
• After putting the thread on the wait queue
• Release puts the thread on the ready queue, but the thread still thinks it needs to go to sleep
• Misses wakeup and still holds lock (deadlock!)
在 sleep 后的话就要让中断它的来 enable 然后 disable 再返回去

L6 Locks, Semaphores

① 原子指令序列 atomic instruction sequence: 这些指令自动读取一个值并写入一个新值•硬件负责正确实现这一点•在两个单处理器上(不太难) •和多处理器(需要缓存一致性协议 cache coherence protocol 的帮助)•不像禁用中断，可以在单处理器和多处理器上使用
② Atomic Read-Modify-Write Instructions (textbook spin-lock *P127*)

```
• test&set (&address) {           /* most architectures */
      result = M[address];        // return result from "address" and
      M[address] = 1;             // set value at "address" to 1
      return result;
  }
• swap (&address, register) {     /* x86 */
      temp = M[address];          // swap register's value to
      M[address] = register;      // value at "address"
      register = temp;
  }
• compare&swap (&address, reg1, reg2) { /* 68000 */
      if (reg1 == M[address]) {   // If memory still == reg1,
          M[address] = reg2;      // then  put reg2 => memory
          return success;
      } else {                    // Otherwise do not change memory
          return failure;
      }
  }
• load-linked&store-conditional(&address) { /* R4000, alpha */
      loop:
          ll r1, M[address];
          movi r2, 1;             // Can do arbitrary computation
          sc r2, M[address];
          beqz r2, loop;
  }
addToQueue(&object) {
   do {                      // repeat until no conflict
      ld r1, M[root]         // Get ptr to current head
      st r1, M[object]       // Save link in new object
   } until (compare&swap(&root,r1,object));
}
```

Busy-Waiting: thread consumes cycles while waiting.
For multiprocessors: every test&set() is a write, which makes value ping-pong around in cache (using lots of network BW)
Priority Inversion **优先级翻转**: If busy-waiting thread has higher priority than thread holding lock.
Multiprocessor Spin Locks: test&test&set:
Int mylock=0; /*Free*/
Acquire(){do{while(mylock); /*Wait until might be free*/}while(test&set (&mylock)); /*exit if get lock*/}
Release() {mylock= 0;}
③ ***Semaphore*** has a non-negative integer value and supports the following two operations:
P(): an atomic operation that waits for semaphore to become positive, then decrements it by 1
V(): an atomic operation that increments the semaphore by 1, waking up a waiting P, if any
Semaphore 的两个用法:互斥(initial value=1): P()和 V()中塞 critical section. Scheduling Constraints (initial value = 0): Allow thread 1 to wait for a signal from thread 2 • thread 2 schedules thread 1 when a given event occurs
Use a separate semaphore for each constraint 约束
④ ***Monitor***: a lock and zero or more condition variables for managing concurrent access to shared data - Use locks for mutual exclusion and condition variables for scheduling constraints
⑤ ***Condition Variable***: a queue of threads waiting for something inside a critical section
Key idea: make it possible to go to sleep inside critical section by atomically releasing lock at time we go to sleep
和 semaphores 不同: Can't wait inside critical section

```
Lock lock;
Condition dataready;
Queue queue;

AddToQueue(item) {
   lock.Acquire();              // Get Lock
   queue.enqueue(item);         // Add item
   dataready.signal();          // Signal any waiters
   lock.Release();              // Release Lock
}

RemoveFromQueue() {
   lock.Acquire();              // Get Lock
   while (queue.isEmpty()) {
      dataready.wait(&lock); // If nothing, sleep
   }
   item = queue.dequeue();      // Get next item
   lock.Release();              // Release Lock
   return(item);
}
```

L7 Deadlock and Starvation

① Resources: Serially reusable resources (CPU cycles, memory space, I/O devices, files); Consumable resources (message, buffer of information, interrupts)
② Conditions for Deadlock: Mutual exclusion; Hold and wait; No preemption 抢占; Circular wait.
③ Resource Allocation Graph:
− If graph contains no cycles: No Deadlock
− If graph contains a cycle: If only one instance per resource type, then deadlock; If several instances per resource type, possibility of deadlock

④ How to deal with deadlocks?

－Deadlock prevention (*P181*): Ensure that at least one of the necessary conditions for deadlock cannot hold.

－Deadlock avoidance (*6.5.4 P183* Banker's algorithm): Provide advance information to the operating system on which resources a process will request throughout its lifetime.

－Deadlock detection (*P188*): Allow deadlocks to occur but then rely on the operating system to detect the deadlock and deal with it.

－Ignore deadlocks: By far the most common approach is simply to let the user deal any potential deadlock.

⑤ Resource allocation state

－# of available and allocated resources

－the maximum demands of the processes

⑥ Starvation: Process waits indefinitely

Deadlock: Circular waiting for resources

L8 Bound-buffer, Reader/Writer, Dining Philosopher

① Bounded-Buffer Problem

Producer process (creates filled buffers)

```
do {
…
/*produce an item in next_produced*/
…
wait (empty);
wait (mutex);
…
/*add next produced to the buffer*/
…
signal(mutex);
signal(full);
} while (true);
```

Consumer process (empties filled buffers)

```
do {
wait (full);
wait (mutex);
…
/*remove an item from to the
next_consumed*/
…
signal(mutex);
signal(empty);
…
/*consume the item in next consumed*/
…
} while (true);
```

② Reader/Writer Problem:

同时可以多个读者只能一个作者

• Reader():Wait until no writers //Access data base//Check out – wake up a waiting writer

• Writer():Wait until no active readers or writers//Access database //Check out – wake up waiting readers or writer

• State variables (Protected by a lock called "lock"):

• int AR: # of active readers; initially = 0

• int WR: # of waiting readers; initially = 0

• int AW: # of active writers; initially = 0

• int WW: # of waiting writers; initially = 0

• Condition okToRead= NIL

• Condition okToWrite= NIL

用 okContinue 来代替 okToRead 和 okToWrite，必须用 broadcast 代替 signal，读者作者都会在这个变量睡着

```
Reader() {
  // First check self into system
  lock.Acquire();

  while ((AW + WW) > 0) { // Is i
    WR++;                 // No. '
    okToRead.wait(&lock); // Slee
    WR--;                 // No l
  }

  AR++;                   // Now '
  lock.release();

  // Perform actual read-only acc
  AccessDatabase(ReadOnly);

  // Now, check out of system
  lock.Acquire();
  AR--;                   // No l
  if (AR == 0 && WW > 0)  // No o
    okToWrite.signal();   // Wake
  lock.Release();
}
```

```
Writer() {
  // First check self into system
  lock.Acquire();

  while ((AW + AR) > 0) { // Is it
    WW++;                  // No. A
    okToWrite.wait(&lock); // Sl
    WW--;                  // No lo
  }

  AW++;                    // Now w
  lock.release();

  // Perform actual read/write ac
  AccessDatabase(ReadWrite);

  // Now, check out of system
  lock.Acquire();
  AW--;                    // No lo
  if (WW > 0){             // Give
    okToWrite.signal();    // Wake
  } else if (WR > 0) {     // Other
    okToRead.broadcast();  // Wake
  }
  lock.Release();
}
```

③ Monitor Solution to Dining Philosophers

```
monitor DiningPhilosophers {
        enum {THINKING, HUNGRY, EATING} state[5];
        condition self[5];
        void pickup (int i){
             state[i] = HUNGRY;
             test(i); /*test left and right are not eating*/
             if (state[i] != EATING) self[i].wait;
        }
        void putdown (int i){
             state[i] = THINKING;
             /*test left and right neighbors*/
               test ((i + 4) % 5); /*signal on neighbor*/
               test ((i + 1) % 5); /*signal other neighbor*/
        }
        void test (int i) {
             if ((state[(i + 4) % 5] != EATING)) &&
             (state[i] == HUNGRY)&&
             (state[(i + 1) % 5] != EATING){
                     state[i] = EATING;
             self.signal();
             }
        }
        initialization_code() {
             for (int i = 0; i < 5; i++)
             state[i] = THINKING;
        }
}
```

52

•Resource hierarchy solution

Take the lower priority resource first and take the higher priority resource later; when finished, put the higher first.

•Arbitrator solution

Need have permission for picking up the forks.

•Chandy/Misra solution

Send request, if the philosopher has finished (dirty forks), clean forks and give it to the request philosopher.

④ 怎么用 sema 来构造 monitor？*5.8 P136*

L9 Scheduling

① Scheduling Assumption:

• One program per user

• One thread per program

• Programs are independent

高级目标:分配 CPU 时间来优化系统的某些期望参数

② Assumption: CPU Bursts:

Execution model: programs alternate between bursts of CPU(compute) and I/O

③ Scheduling Policy Goals/Criteria:

a. Minimize Response Time

b. Maximize Throughput

-Maximize operations/ jobs per second

-Minimizing response time will lead to more context switching than if you only maximized throughput

-Two parts to maximizing throughput

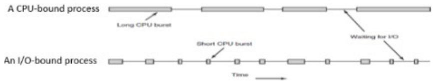
 > Minimize overhead (e.g., context-switching)

 > Efficient use of resources (CPU, disk, memory, etc.)

c. Fairness

Share CPU among users in some equitable way.

④ Program 一会儿用 CPU 一会儿用 I/O• Each scheduling decision is about which job to give to the CPU for use by its next CPU burst •通过 time slicing，thread 可能在完成当前 CPU burst 前被强行放弃 CPU。有些 I/O 活动可以算作计算。例如，当 CPU 将位复制到视频 RAM 以更新屏幕时，它是在计算，而不是执行 I/O，因为 CPU 正在使用中。在这种意义上，I/O 是指进程进入阻塞状态，等待外部设备完成其工作。



调度策略目标/标准

- 最小化回应时间 Minimize Response Time
  - 最小化执行操作(或工作)所需的时间
  - 回应时间是用户所能看到的
    - 响应编辑器击键的时间
    - 编译程序的时间
    - 实时任务:必须满足世界规定的最后期限
- 最大化吞吐量 Maximize Throughout
  - 最大化每秒的操作或工作
  - 与响应时间相关的吞吐量，但并不相同:
    - 与只最大化吞吐量相比，最小化响应时间将导致更多的上下文切换
  - 最大化吞吐量的两部分:
    - 最小化开销(例如，上下文切换)
    - 高效使用资源(CPU、磁盘、内存等)。
- 公平Fairness
  - 在用户之间以某种公平的方式共享CPU
  - 公平性并不是最小化平均响应时间:
    - 通过降低系统的公平性来提高平均响应时间

⑤ <u>First-Come, First-Served (FCFS)</u>

－Pros: simple

－Cons: short jobs get stuck behind long ones

<u>Round Robin Scheduling</u>

－Pros: Better for short jobs, Fair.

－Cons: Context switching overhead for long jobs.

Round Robin Scheduling (example)

• Time quantum = 4 ms

• Waiting Time for

 –P1: (10-4) = 6

 –P2: (4-0) = 4

 –P3: (7-0) = 7

Process	Burst Time
P_1	24
P_2	3
P_3	3

P_1	P_2	P_3	P_1	P_1	P_1	P_1	P_1
0–4	4–7	7–10	10–14	14–18	18–22	22–26	26–30

• Average waiting time = (6+4+7)/3 = 5.66ms

• Average completion time = (30+7+10)/3 = 15.67ms

• Pros: Better for short jobs, Fair

• Cons: Context switching overhead for long jobs

<u>Priority Scheduling</u>

问题：starvation:低的因为高的不能跑；死锁:优先级反转 Fix: Dynamic priorities –adjust base-level priority up or down based on heuristics about interactivity, locking, burst behavior; Tradeoff: fairness gained by hurting average response time!

<u>Lottery Scheduling</u>

－Advantage over strict priority scheduling: behaves gracefully as load changes

<u>Shortest Job First (SJF) / Shortest Time to Completion First (STCF)</u>

<u>Shortest Remaining Time First (SRTF) / Shortest Remaining Time to Completion First (SRTCF)</u>

－Pros: Optimal (average response time)

－Cons: Hard to predict future, Unfair

－SJF/SRTF are the best you can do at minimizing average response time

－SRTF can lead to starvation if many small jobs!

⑥ Multilevel Queue (*7.1.5 P211*)

Foreground 前端高优先级 RR，background 后端 FCFS; Job 一开始在最高级队列,如果超时就降一级,没超时升一级

调度在队列间做，固定优先级调度(从最高的开始往下),time slice(每个队列得到固定量的 CPU 时间,70%给最高 20%下一级 10%最低)

时间片粒度 Timeslice Granularity

Linux O(1)scheduler：140 个优先队列，0-99kernel/realtime task,100-139user task 两个队列 active 和 expired 过期；每个优先队列里 FIFO，用完时间就去过期队列

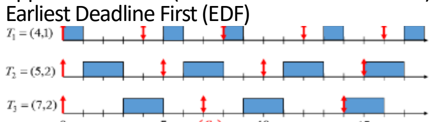
User-task priority adjusted ±5 based on heuristics • p->sleep_avg= sleep_time – run_time 越高越 I/O bound

Interactive credit 睡眠长获得，运行长花费

Linux Completely Fair Scheduler (CFS) 通过权重反映的优先级，例如将任务的权重增加 1，无论当前的优先级如何，CPU 时间的增量都是相同的 红黑树每次运行已运行时长最短的

Real-Time Scheduling (RTS): Hard Real-Time • Attempt to meet all deadlines • EDF (Earliest Deadline First), LLF (Least Laxity First), RMS (Rate-Monotonic Scheduling), DM (Deadline Monotonic Scheduling) • Soft Real-Time • Attempt to meet deadlines with high probability • Minimize miss ratio / maximize completion ratio (firm real-time) • Important for multimedia applications • CBS (Constant Bandwidth Server)

Earliest Deadline First (EDF)



Schedulable when $\sum_{i=1}^{n}\left(\frac{C_i}{P_i}\right) \le 1$

Additional 其他

① TCB 里的东西： Execution State: CPU registers, program counter (PC), pointer to stack (SP) • Scheduling info: state, priority, CPU time • Various Pointers (for implementing scheduling queues) • Pointer to enclosing process (PCB)