Midterm Review

OS Structure

program always runs

- User mode/ kernel mode (Dual-Mode)
  - Memory protection, privileged instructions

Hardware support to distinguish app/kernel
– Privileged instructions are only for kernel mode
– Applications can enter into kernel mode only via pre-defined system calls

- System call 当user 需要与内核交流时候，需要调用提前预定好的system call
  - Definition, examples, how it works?

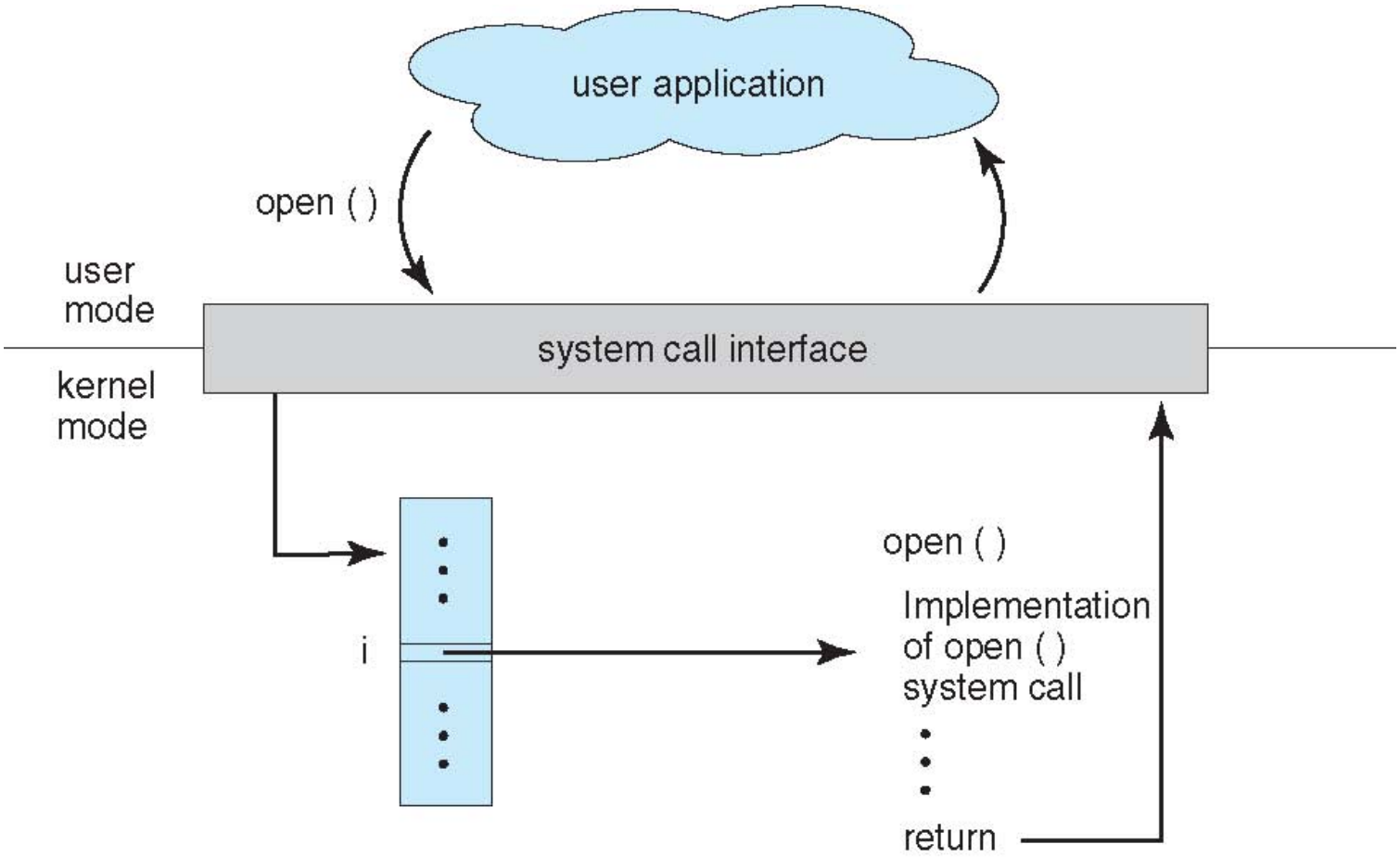
system calls
– Software interrupt. “int <num>” in Intel x86
Programming interface to the services provided by the OS

- Other concepts to know
  - Monolithic kernel vs. Micro kernel

Implements CPU scheduling, memory management, filesystems, and other OS modules all in a single big chunk
Pros and Cons
+ Overhead is low
+ Data sharing among the modules is easy
– Too big.(devicedrivers!!!)
– A bug in one part of the kernel can crash the entire system

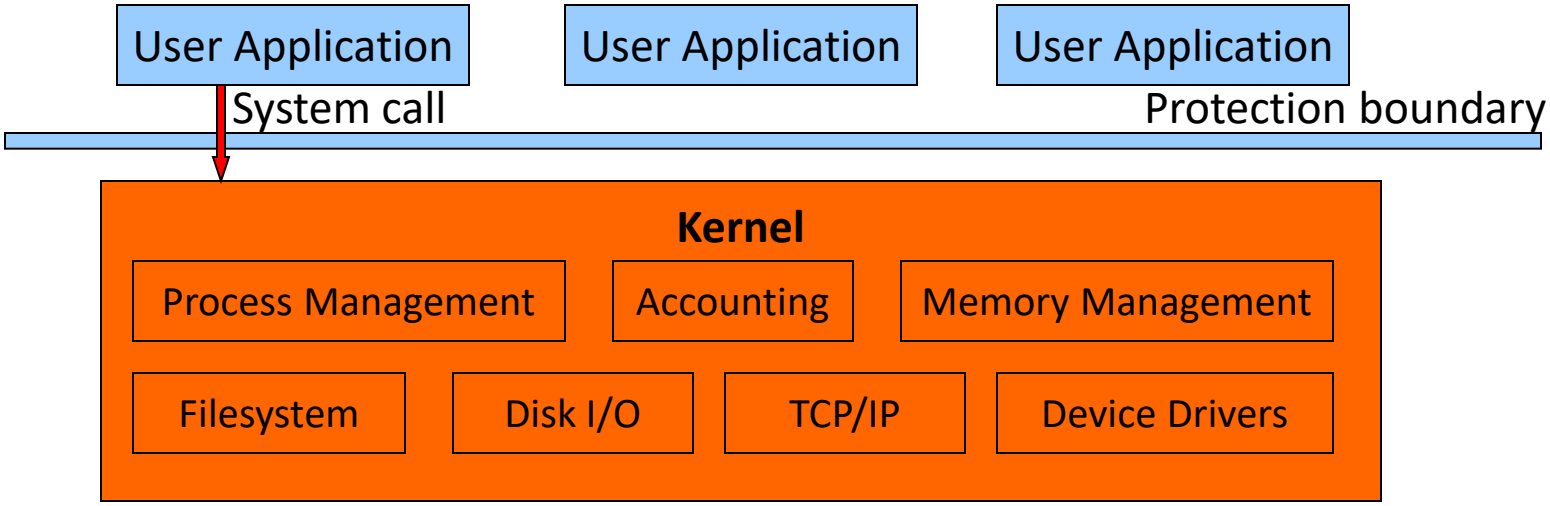
Moves as much from the kernel into user space Communicate among kernels and user via message passing
Pros and Cons
+ Easy to extend (user level driver)
+ More reliable (less code is running in kernel mode)
- Performance overhead of user space to kernel space communication

API - System Call - OS



UNIX: Monolithic Kernel

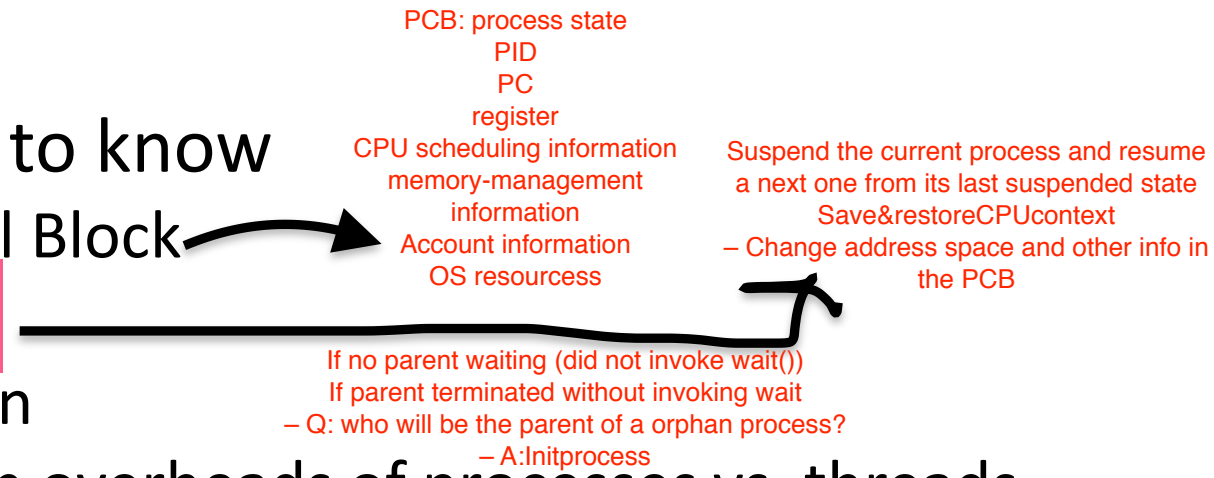
- Implements CPU scheduling, memory management, filesystems, and other OS modules all in a single big chunk



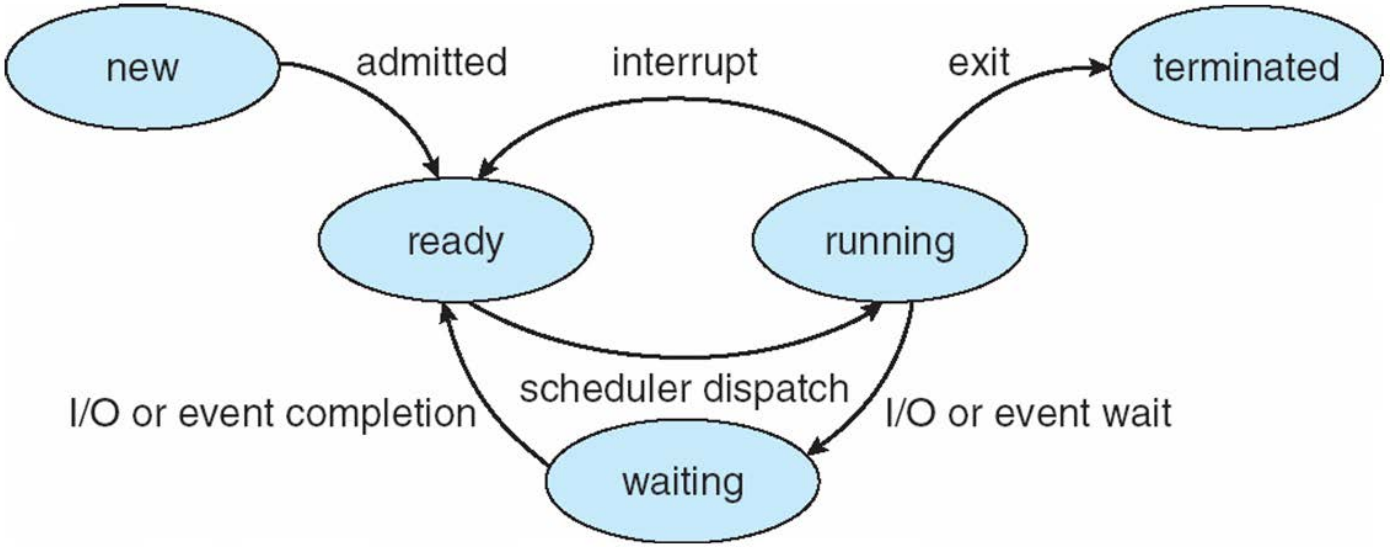
- Pros and Cons
  - + Overhead is low
  - + Data sharing among the modules is easy
  - – Too big. (device drivers!!!)
  - – A bug in one part of the kernel can crash the entire system

Process

- Address space layout
  - Code, data, heap, stack
- Process states
  - new, ready, running, waiting, terminated
- Other concepts to know
  - Process Control Block
  - Context switch
  - Zombie, Orphan
  - Communication overheads of processes vs. threads



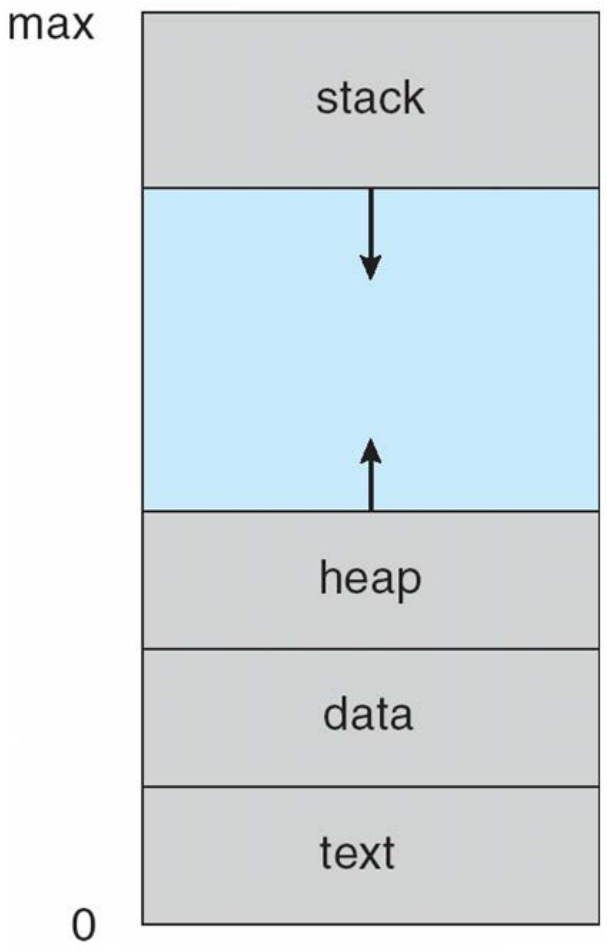
Process State



- **running**: Instructions are being executed
- **waiting**: The process is waiting for some event to occur
- **ready**: The process is waiting to be assigned to a processor

Process Address Space

- Text
  - Program code
- Data
  - Global variables
- Heap
  - Dynamically allocated memory
    - i.e., Malloc()
- Stack
  - Temporary data
  - Grow at each function call



Quiz

```
int count = 0;
int main()
{
   int pid = fork();
   if (pid == 0){
      count++;
      printf("Child: %d\n", count);
   } else{
      wait(NULL);
      count++;
      printf("Parent: %d\n", count);
   }
   count++;
   printf("Main: %d\n", count);
   return 0;
}
```

- Hints
  - Each process has its own private address space
  - Wait() blocks until the child finish

- Describe the output.

  Child: 1

  Main: 2

  Parent: 1

  Main: 2

Inter-Process Communication

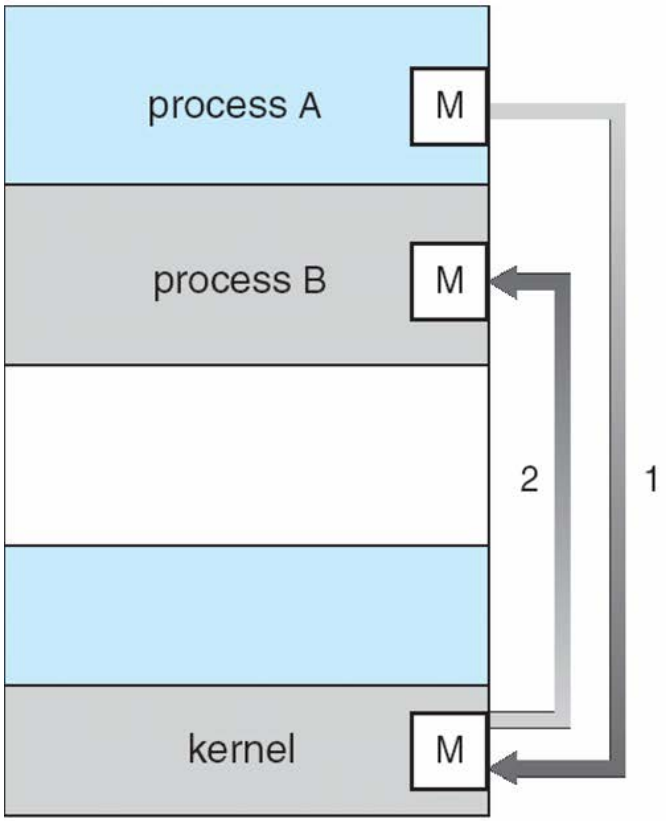
- Shared memory
- Message passing

Shared memory
– share a region of memory between co-operating processes
– read or write to the shared memory region –
++ fast communication
– -- synchronization is very difficult
• Message passing
– exchange messages (send and receive)
– typically involves data copies (to/from buffer) –
++ synchronization is easier
– -- slower communication

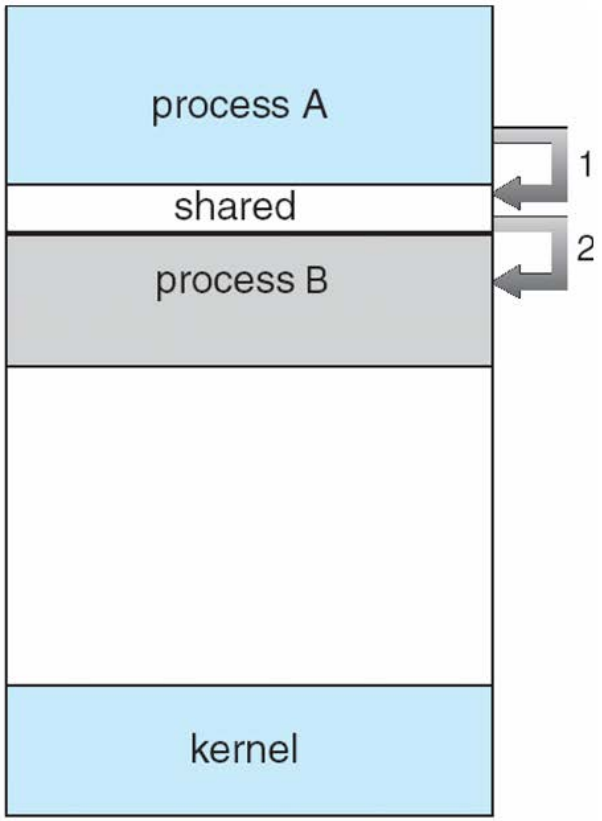
Models of IPC

需要去内核取资源，交流缓慢

在user model 分享资源，但实现同步很困难，基本用来交流



(a)



(b)

message passing

shared memory

Quiz

- A process produces 100MB data in memory. You want to share the data with two other processes so that each of which can access half the data (50MB each). What IPC mechanism will you use and why?

- IPC mechanism: POSIX Shared memory
- Reasons: (1) large data → need high performance, (2) no need for synchronization,

Threads

- User threads vs. Kernel threads
  - user-threading: kernel don't understand threads
- Key benefits over processes?
  + Performance
  - Protection

Multi-threads vs. Multi-processes

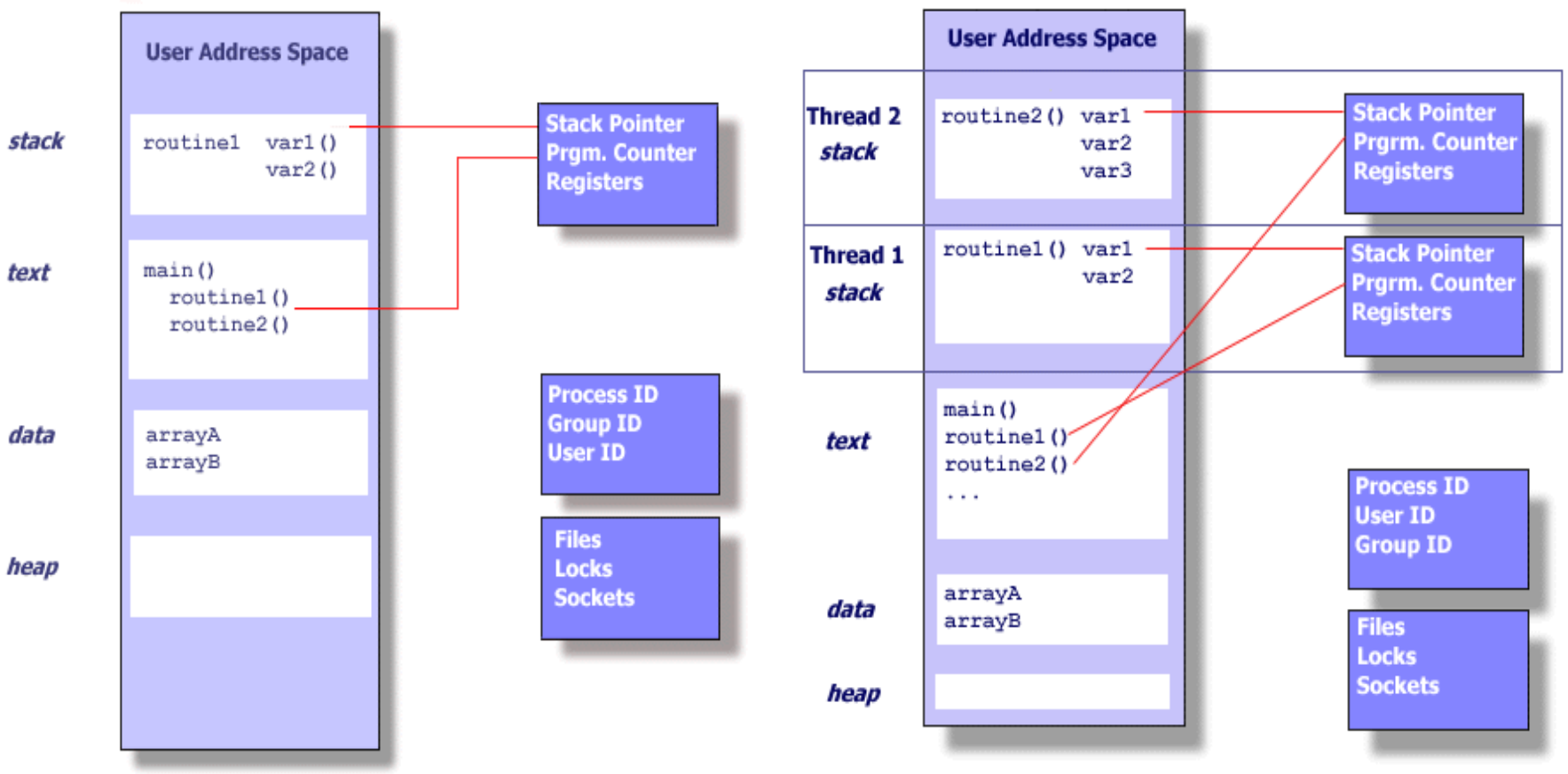
- Multi-processes
  - (+) protection
  - (-) performance (?)

Process-per-tab

- Multi-threads
  - (+) performance
  - (-) protection

Single-process multi-threads

Single and Multithreaded Process



source: https://computing.llnl.gov/tutorials/pthreads/

Synchronization

- Race condition

A situation when two or more threads read and write shared data at the same time

- Synchronization instructions
  - *test&set, compare&swap*
- Spinlock
  - Spin on wait
  - Good for short critical section but can be wasteful
- Mutex
  - Block (sleep) on wait
  - Good for long critical section but bad for short one

Race Condition

Initial condition: *counter = 5*

Thread 1	Thread 2
R1 = load (counter);	**R2 = load (counter);**
R1 = R1 + 1;	**R2 = R2 – 1;**
counter = store (R1);	**counter = store (R2);**

- What are the possible outcome?

5, 4, 6

Quiz

- Write the pseudo-code definition of *TestAndSet* instruction

```
Int TestAndSet(int *lock)
{
   int ret;
   ____________________
   ____________________
   return ret;
}
```

ret = *lock
*lock = True

- Complete the following lock implementation using *TestAndSet* instruction

```
void init_lock(int *mutex)
{
   *mutex = 0;
}
void lock(int *mutex)
{
   while (____________________);
}
void unlock(int *mutex)
{
   ____________________
}
```

TestAndSet(mutex)

*metex = 0

Quiz

- Pseudo code of test&set

```
int TestAndSet(int *lock) {
  int ret = *lock;
  *lock = 1;
  return ret;
}
```

```
int CAS(int *value, int oldval, int newval) {
    int temp = *value;
    if (*value == oldval)
        *value = newval;
    return temp;
}
```

- Spinlock implementation using *TestAndSet* instruction

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

void lock(int *lock) {
  while (TestAndSet(lock));
}

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

```
void mutex_init (mutex_t *lock)
{
   lock->value = 0;
   list_init(&lock->wait_list);
   spin_lock_init(&lock->wait_lock);
}

void  mutex_lock (mutex_t *lock)
{
...
   while(TestAndSet(&lock->value)) {
      current->state = WAITING;
      list_add(&lock->wait_list, current);
...
      schedule();
...
   }
...
}

void mutex_unlock (mutex_t *lock)
{
...
   lock->value = 0;
   if (!list_empty(&lock->wait_list))
      wake_up_process(&lock->wait_list)
...
}
```

More reading: mutex.c in Linux

```
void mutex_init (mutex_t *lock)
{
   lock->value = 0;
   list_init(&lock->wait_list);
   spin_lock_init(&lock->wait_lock);
}

void  mutex_lock (mutex_t *lock)
{
   spin_lock(&lock->wait_lock);
   while(TestAndSet(&lock->value)) {
      current->state = WAITING;
      list_add(&lock->wait_list, current);
      spin_unlock(&lock->wait_lock);
      schedule();
      spin_lock(&lock->wait_lock);
   }
  spin_unlock(&lock->wait_lock);
}

void mutex_unlock (mutex_t *lock)
{
   spin_lock(&lock->wait_lock);
   lock->value = 0;
   if (!list_empty(&lock->wait_list))
      wake_up_process(&lock->wait_list)
   spin_unlock(&lock->wait_lock);
}
```

比spinlock更加有优势，睡眠或者等待，而不是无限循环

对waiting list进行加锁

Thread waiting list

To protect waiting list

Thread state change

Add the current thread to the waiting list

Sleep or schedule another thread

Someone is waiting for the lock

Wake-up a waiting thread

Bounded Buffer Problem Revisit

Monitor version

```
Mutex lock;
Condition full, empty;

produce (item)
{
    lock.acquire();
    while (queue.isFull())
        empty.wait(&lock);
    queue.enqueue(item);
    full.signal();
    lock.release();
}

consume()
{
    lock.acquire();
    while (queue.isEmpty())
        full.wait(&lock);
    item = queue.dequeue(item);
    empty.signal();
    lock.release();
    return item;
}
```

queue满了，需要等待空

当queue为空，不应该dequeue，所以等待full

Semaphore version

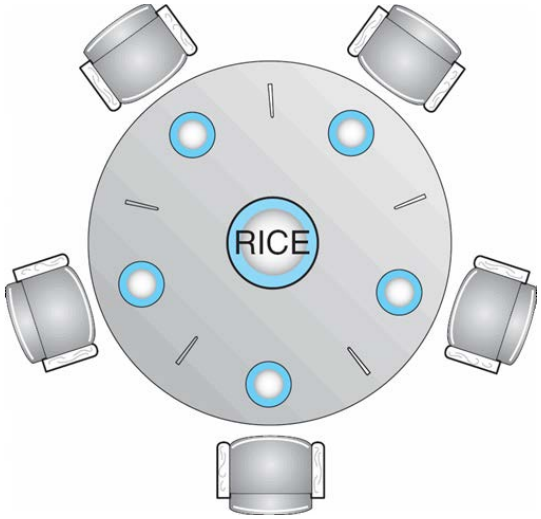
```
Semaphore mutex = 1, full = 0,
empty = N;

produce (item)
{
    empty.P();
    mutex.P();
    queue.enqueue(item);
    mutex.V();
    full.V();
}

consume()
{
    full.P();
    mutex.P();
    item = queue.dequeue();
    mutex.V();
    empty.V();
    return item;
}
```

Deadlock

DeadLock需要避免进程之间互相要
锁然后都进入了等待睡眠的状态

- Deadlock conditions
- Resource allocation graph
- Banker's algorithm
- Dining philosopher example
- Other concepts to know
  - Starvation vs. deadlock

Starvation 饿死的场景很直接，避免资源的不公平利用 ,例如, 不能某些任务
总是获得资源 , 有些任务即使长期在等待下 , 却没有被分配到资源。



Conditions for Deadlocks

- Mutual exclusion
  - only one process at a time can use a resource
- No preemption
  - resources cannot be preempted, release must be voluntary

非抢占；自愿释放资源；非抢占（Non-Preemptive）内核中如果任务不主动放弃，就不会被其他的高优先极任务抢掉对于CPU的控制

- Hold and wait
  - a process must be holding at least one resource, and waiting to acquire additional resources held by other processes
- Circular wait
  - There must be a circular dependency. For example, A waits B, B waits C, and C waits A.

- **All four conditions must simultaneously hold**

Quiz

- Determine whether this state is safe or unsafe.

Total resources: 12

Avail resources: 3

Process	Max	Alloc
P_0	10	4
P_1	3	1
P_2	6	4

10 – 4 <= 8

3 – 1 <= 7

6 – 4 <= 3

Safe

Quiz

- Can a request 3 by P0 be granted?

Total resources: 12

Avail resources: 0

Process	Max	Alloc
P_0	10	7
P_1	3	1
P_2	6	4

$10 - 7 \not\leq 0$

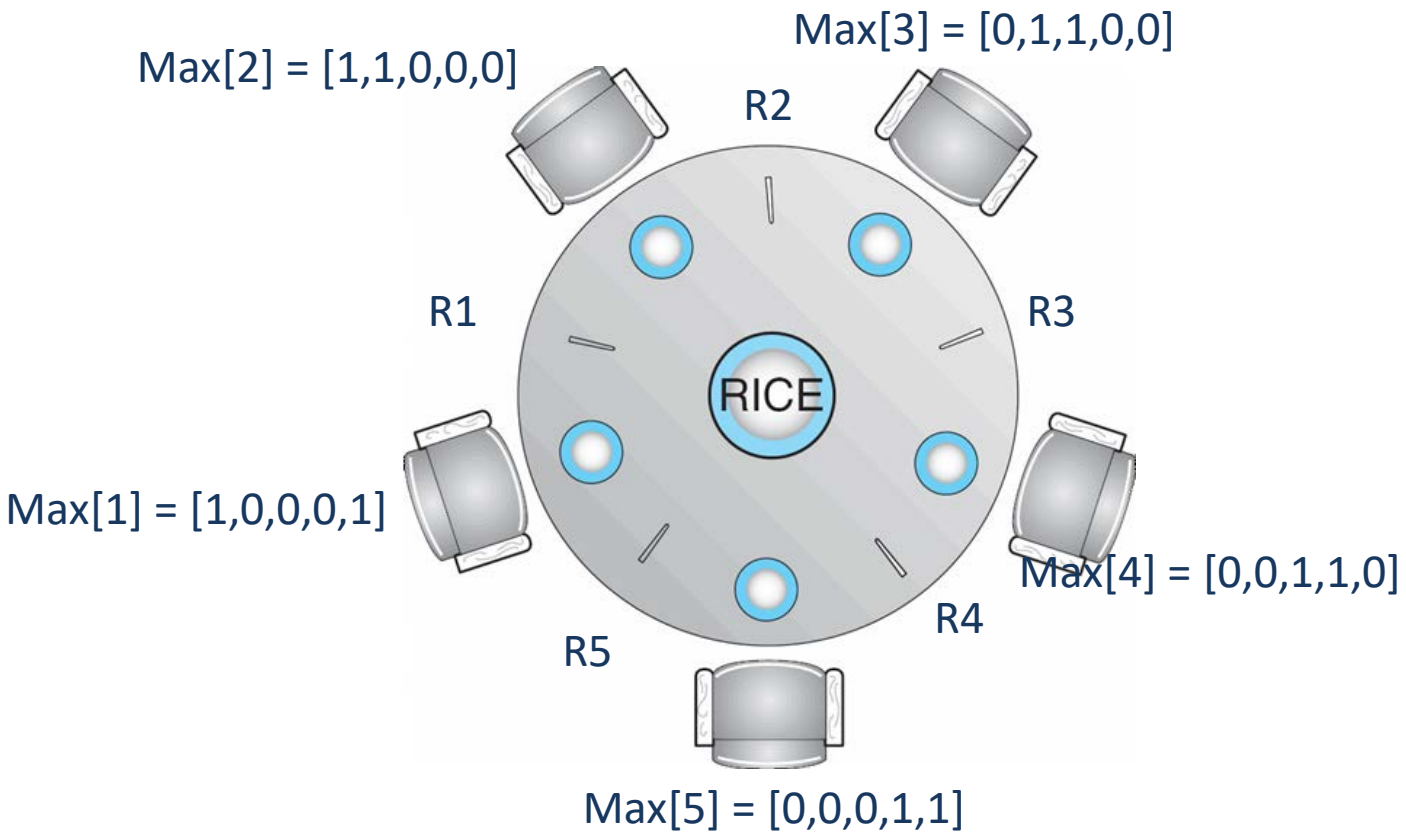
$3 - 1 \not\leq 0$

$6 - 4 \not\leq 0$

Unsafe

Example

Free = [1,1,1,1,1]

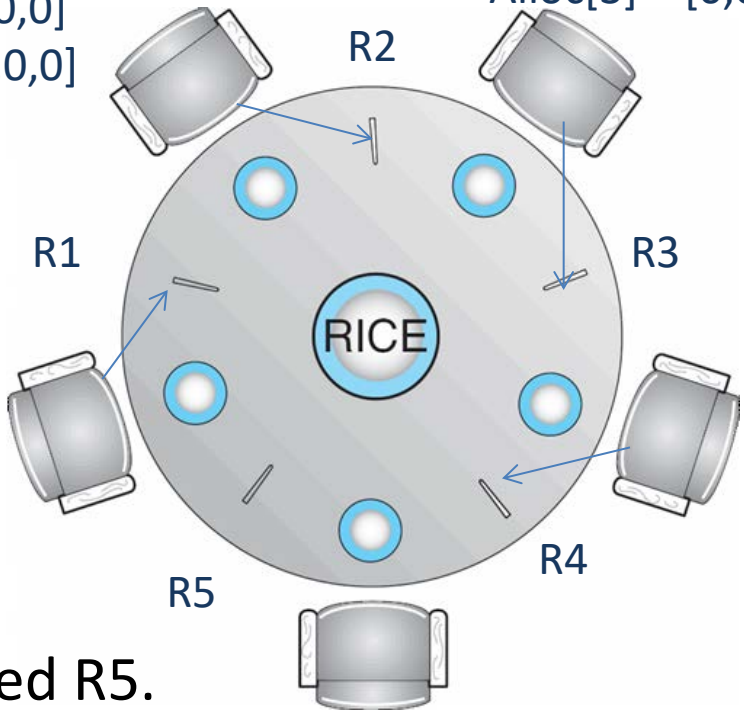


Example

Free = [0,0,0,0,1]
Avail = [0,0,0,0,1]

Max[3] = [0,1,1,0,0]
Alloc[3] = [0,0,1,0,0]

Max[2] = [1,1,0,0,0]
Alloc[2] = [0,1,0,0,0]



Max[1] = [1,0,0,0,1]
Alloc[1]= [1,0,0,0,0]

Max[4] = [0,0,1,1,0]
Alloc[4] = [0,0,0,1,0]

Max[5] = [0,0,0,1,1]
Alloc[5] = [0,0,0,0,0]

- Philosopher 5 requested R5.
- Safe or Unsafe?

Example

Free = [0,0,0,0,0]

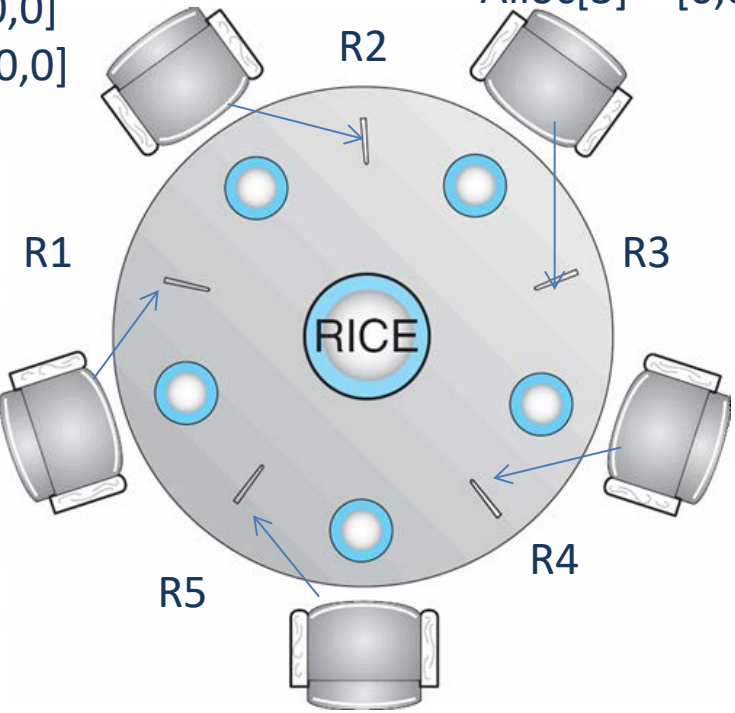
Avail = [0,0,0,0,0]

Max[3] = [0,1,1,0,0]

Alloc[3] = [0,0,1,0,0]

Max[2] = [1,1,0,0,0]

Alloc[2] = [0,1,0,0,0]



Max[1] = [1,0,0,0,1]

Alloc[1]= [1,0,0,0,0]

Max[4] = [0,0,1,1,0]

Alloc[4] = [0,0,0,1,0]

Max[5] = [0,0,0,1,1]

Alloc[5] = [0,0,0,0,1]

2. Find an index i such that

   Finish[i] == false AND

   Max[i] – Alloc[i] ≤ Avail

   If no such i exists, go to step 4

Scheduling

- Three main schedulers
  - FCFS, SJF/SRTF, RR
  - Gant chart examples

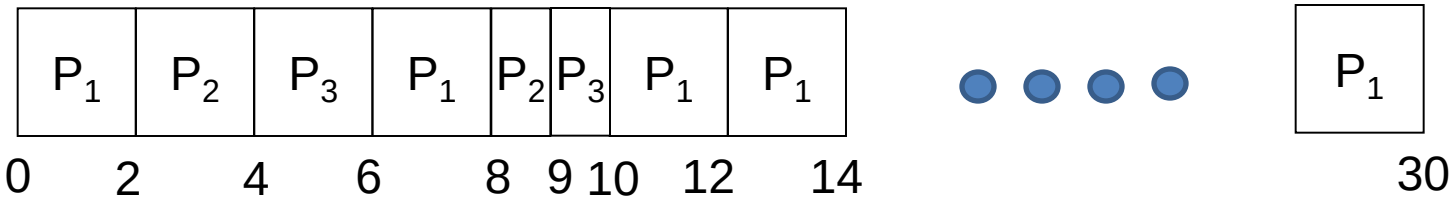
- Other concepts to know ？？
  - Fair scheduling (CFS) not in text? 这块知识没学过！！大纲用以前的
  - Fixed priority scheduling
  - Multi-level queue scheduling
  - Load balancing and multicore scheduling

Round-Robin (RR)

- Example
  - Quantum size = 2

Process	Burst Times
P1	24
P2	3
P3	3

  - Gantt chart



  - **Waiting time**
    - P1: 6, P2: 6, P3: 7. average waiting time = (6+6+7)/3 = 6.33
  - **Sched. latency** (between ready to first schedule)
    - P1: 0, P2: 2, P3: 4. average sched. latency = (0+2+4)/3 = 2