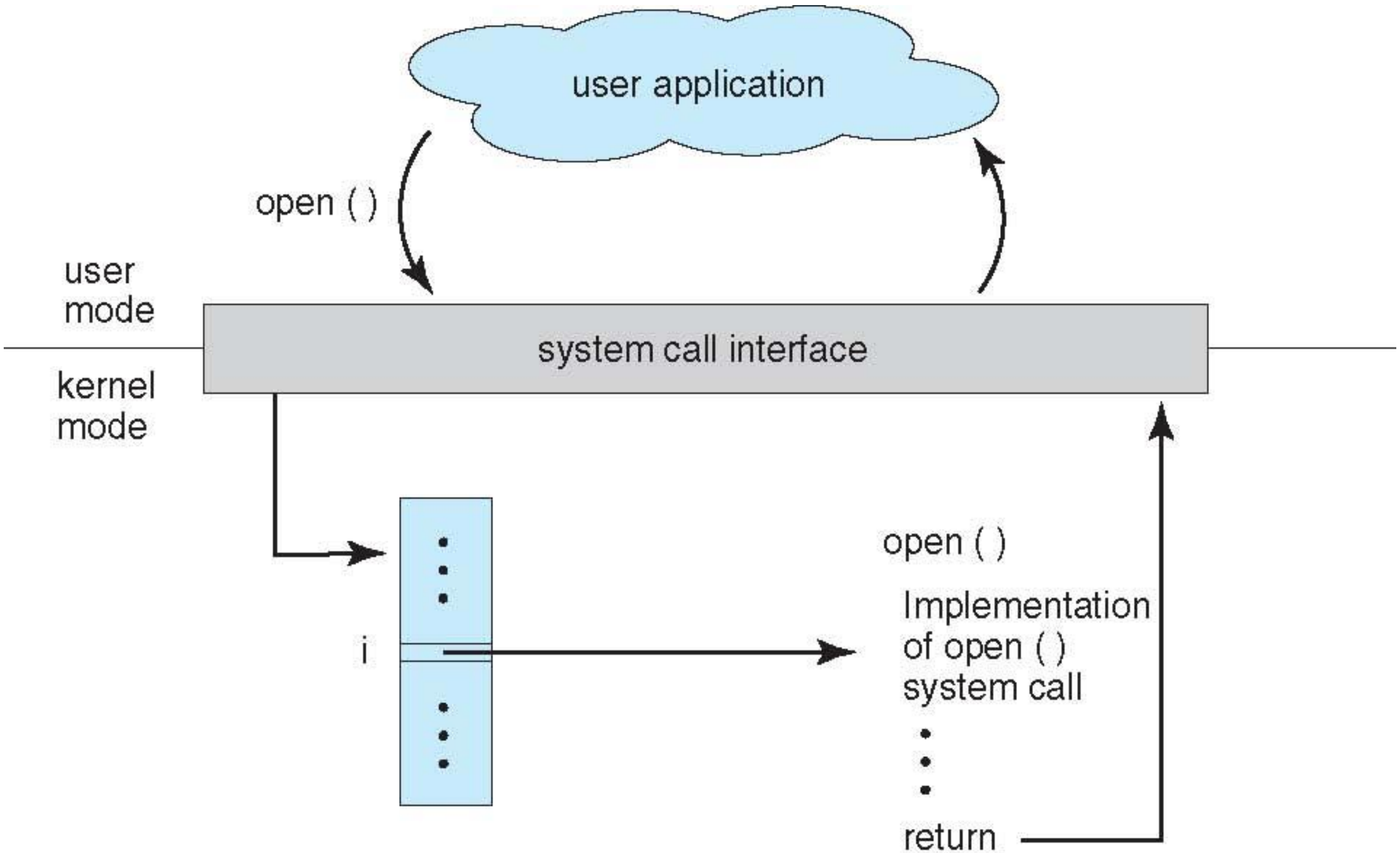
Midterm Review

OS Structure

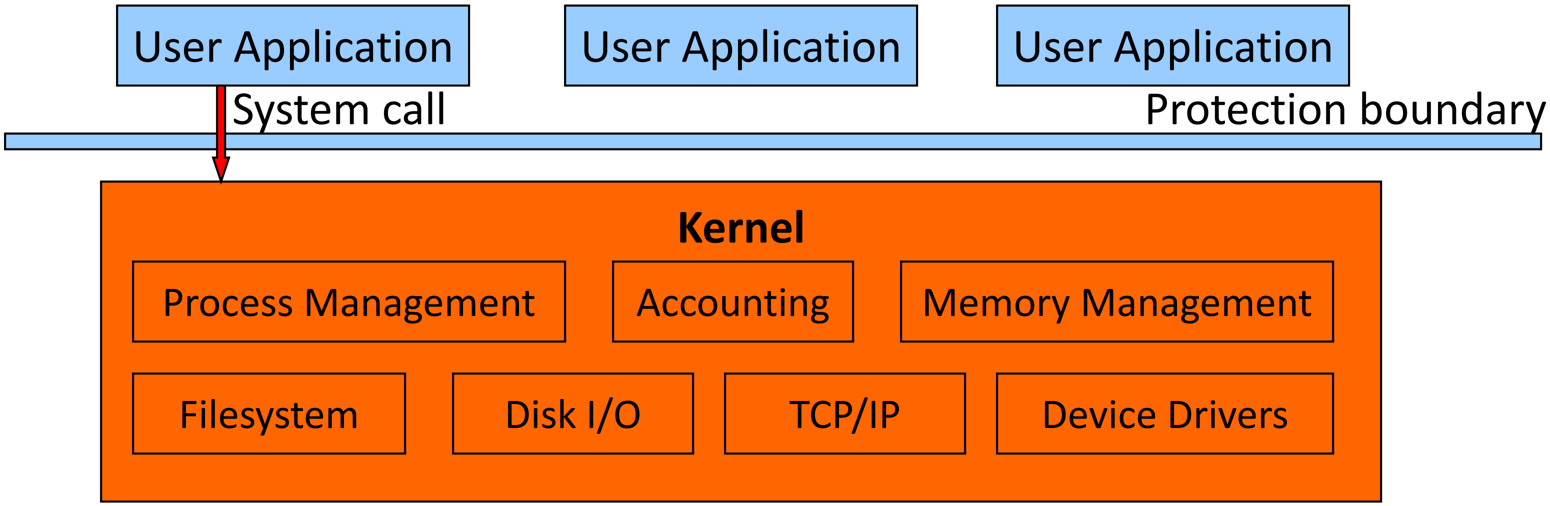
- User mode/ kernel mode
  - Memory protection, privileged instructions
- System call
  - Definition, examples, how it works?
- Other concepts to know
  - Monolithic kernel vs. Micro kernel

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

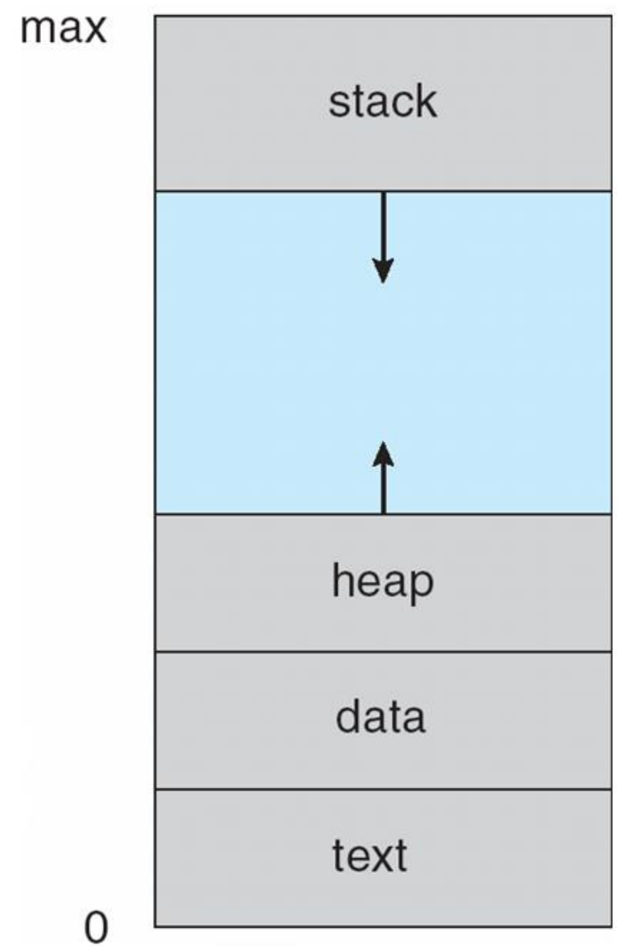
Process

- Address space layout
  - Code, data, heap, stack
- Process states
  - new, ready, running, waiting, terminated

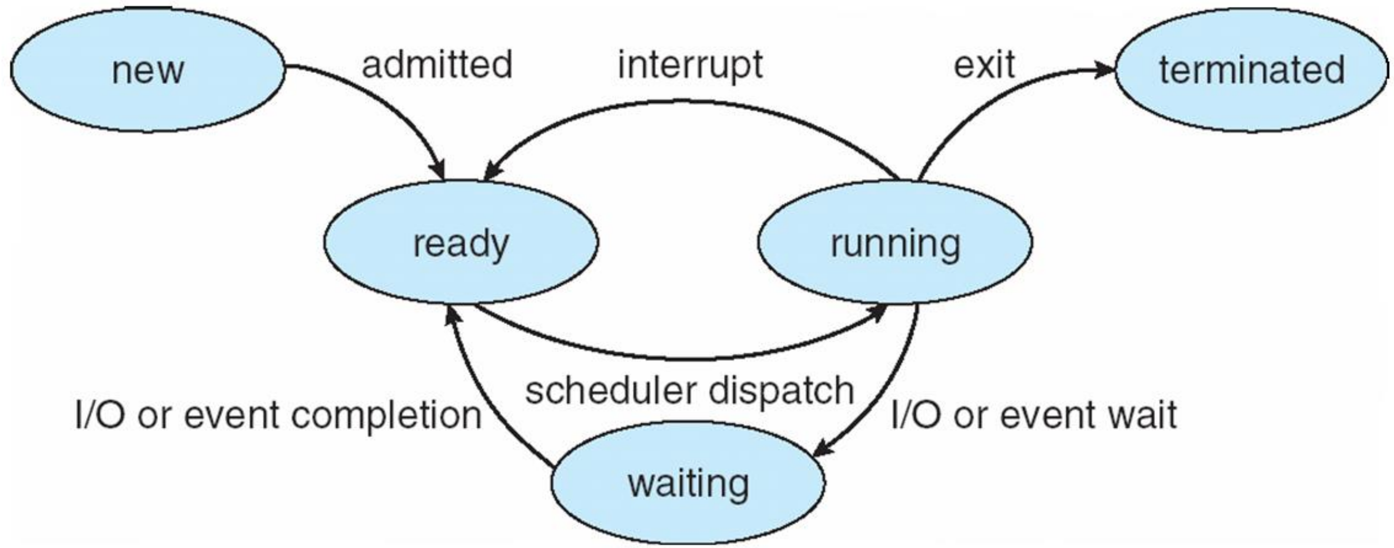
- Other concepts to know
  - Process Control Block
  - Context switch
  - Zombie, Orphan
  - Communication overheads of processes vs. threads

Process Address Space

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



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

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

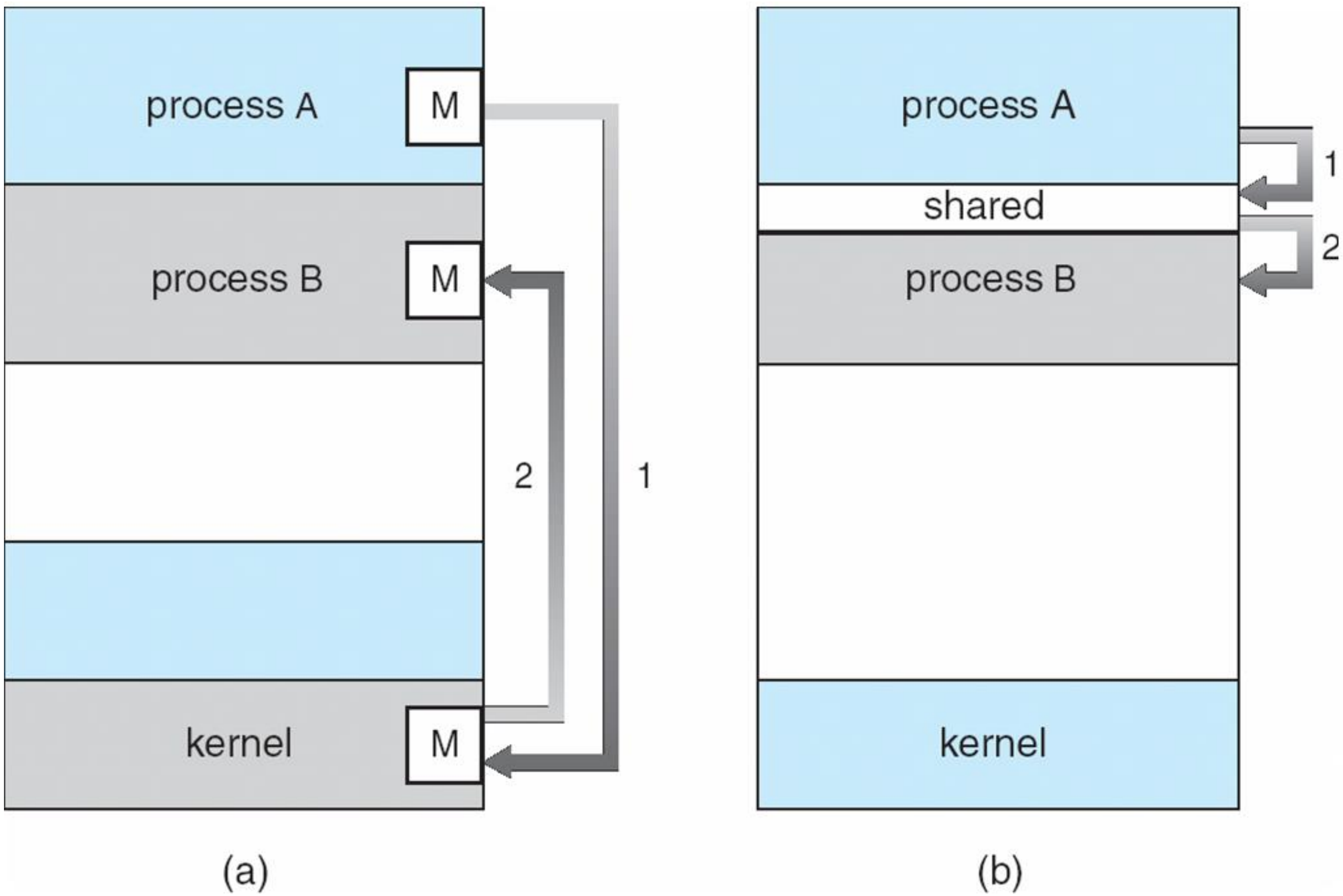
  Parent: 1

  Main: 2

Inter-Process Communication

- Shared memory
- Message passing

Models of IPC



message passing

shared memory

Quiz

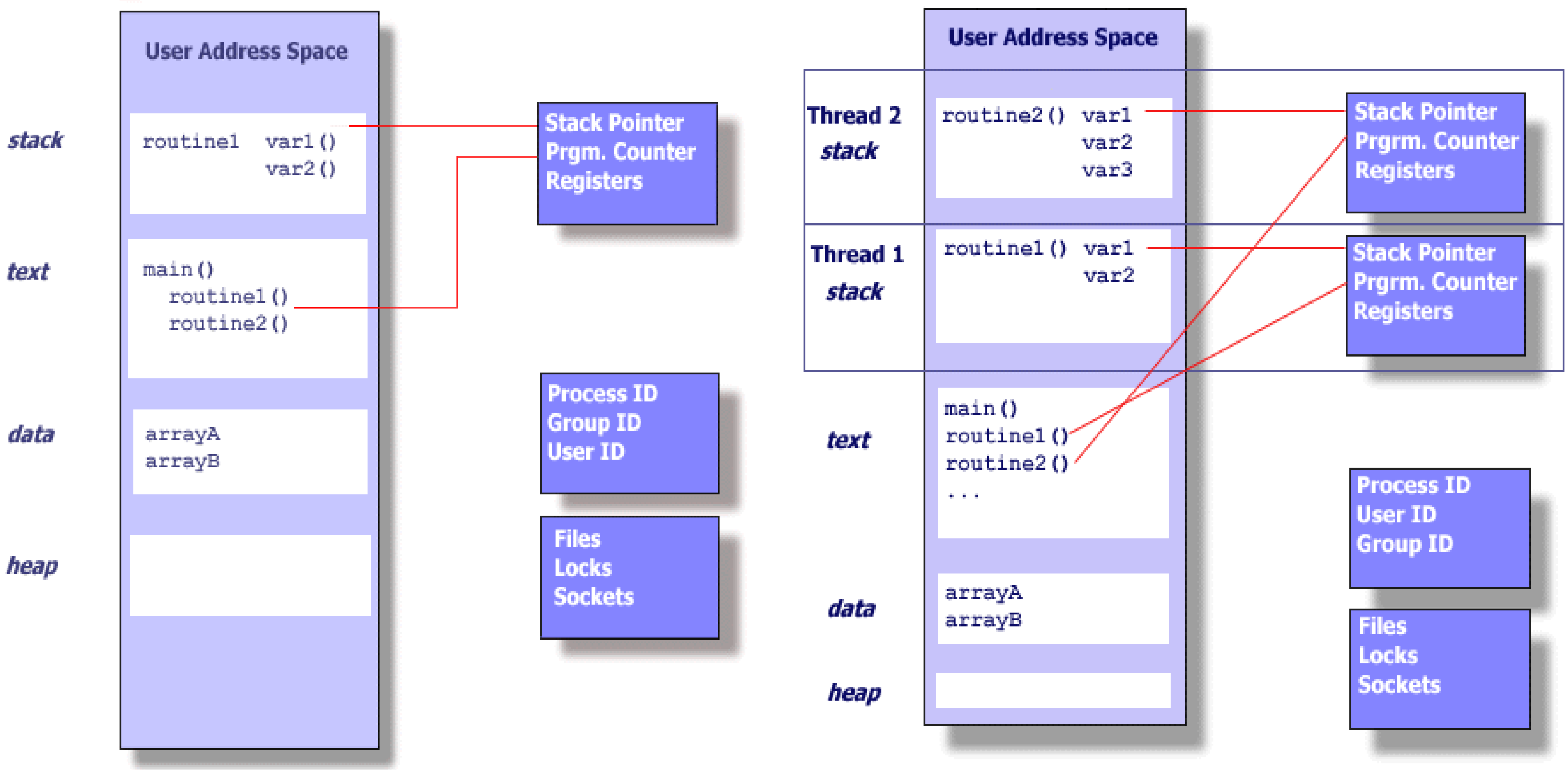
- A process produces 100MB data in memory. You want to share the data with two other processes so that each of which can access half the data (50MB each). What IPC mechanism will you use and why?

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

Threads

- Definition
- Key differences compared to process
- Communication between the threads
- User threads vs. Kernel threads
- Key benefits over processes?

Single and Multithreaded Process



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

Recap: Multi-threads vs. Multi-processes

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

Process-per-tab

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

Single-process multi-threads

Synchronization

- Race condition
- 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

R1 = load (counter);
R1 = R1 + 1;
counter = store (R1);

Thread 2

R2 = load (counter);
R2 = R2 – 1;
counter = store (R2);

- What are the possible outcome?

Quiz

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

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

- Complete the following lock implementation using *TestAndSet* instruction

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

Quiz

- Pseudo code of test&set

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

- 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);
}
```

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

Recap: 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;
}
```

Semaphore version

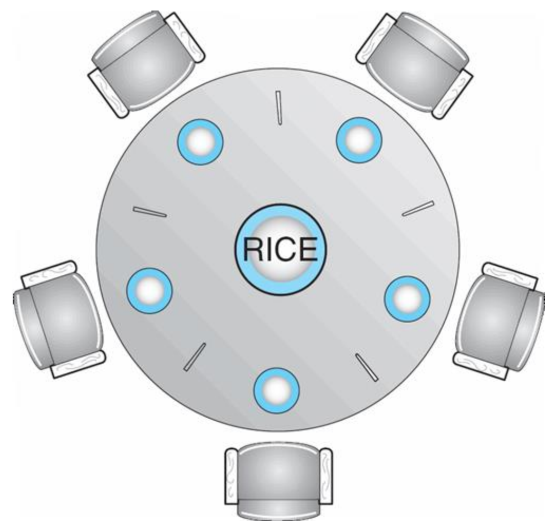
```
Semaphore mutex = 1, full = 0,
empty = N;

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



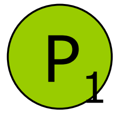
Conditions for Deadlocks

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

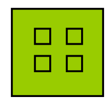
- **All four conditions must simultaneously hold**

Resource-Allocation Graph

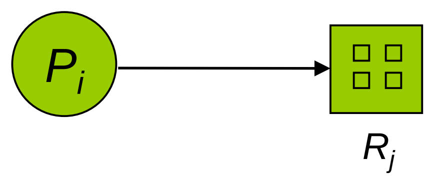
- Process



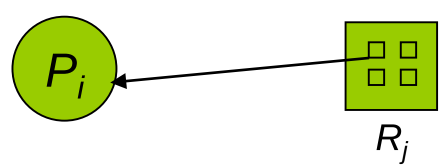
- Resource Type with 4 instances

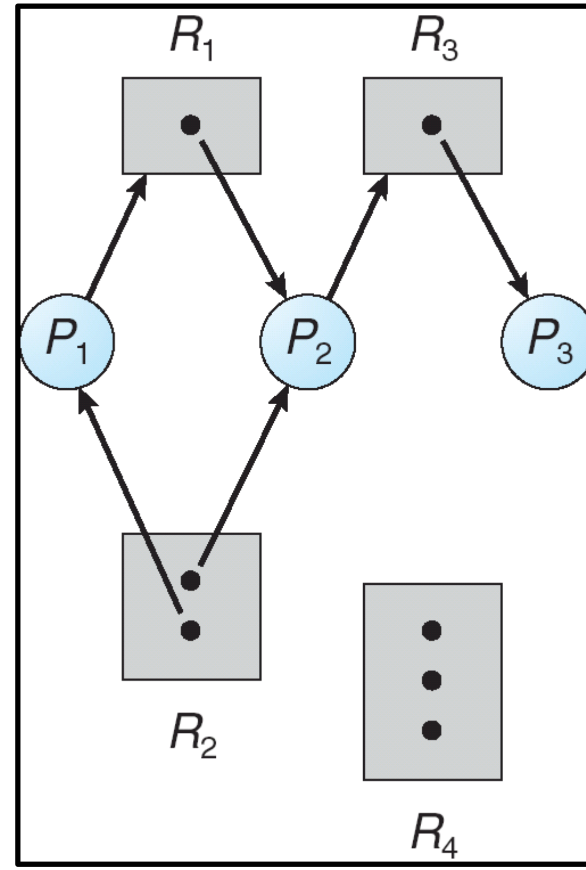


- P_i requests instance of R_j



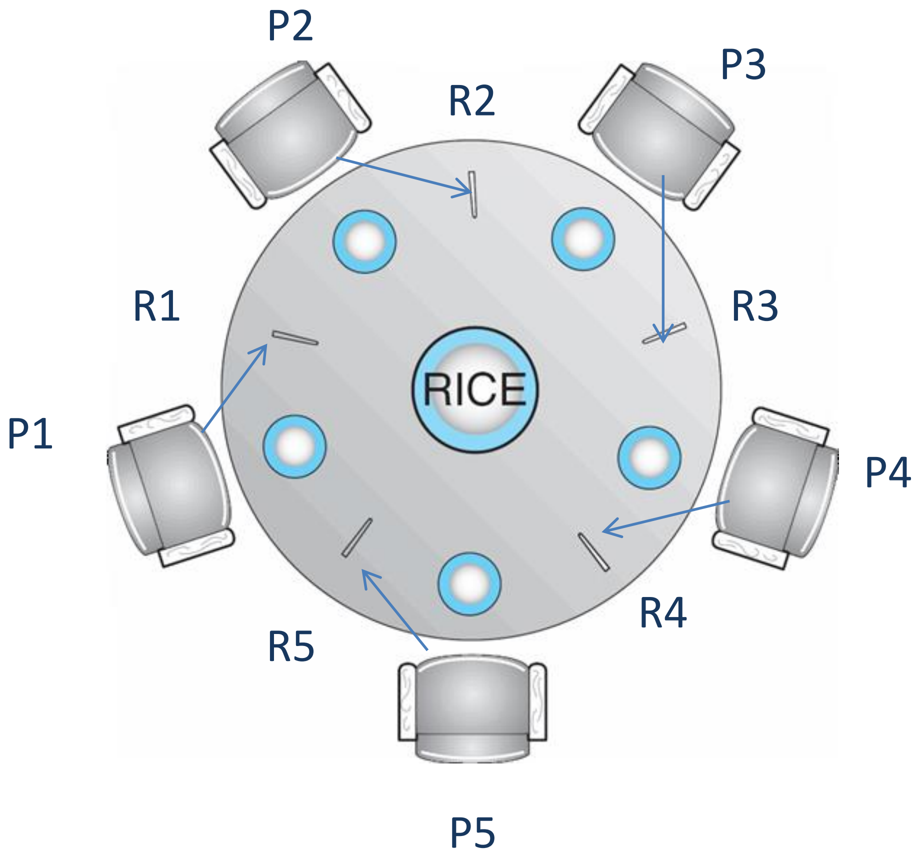
- P_i is holding an instance of R_j





Quiz

Draw a resource allocation graph for the following.



Quiz

- Using Banker's algorithm, determine whether this state is safe or unsafe.

Total resources: 10

Avail resources: 1

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

Scheduling

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

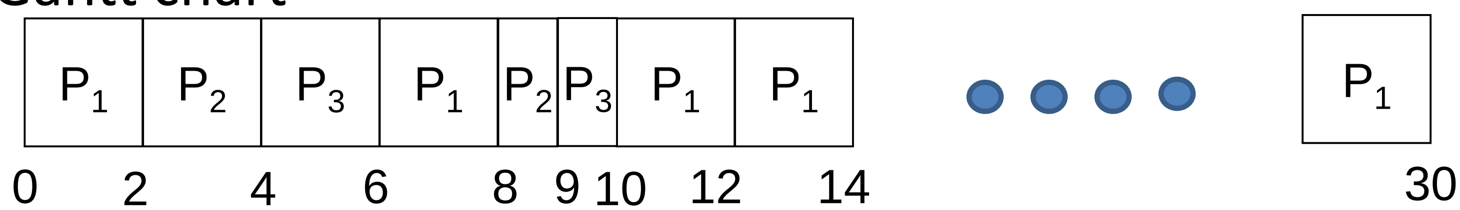
- Other concepts to know
  - Fair scheduling (CFS)
  - Fixed priority scheduling
  - Multi-level queue scheduling
  - Load balancing and multicore scheduling

Round-Robin (RR)

- Example
  - Quantum size = 2

Process	Burst Times
P1	24
P2	3
P3	3

  - Gantt chart



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