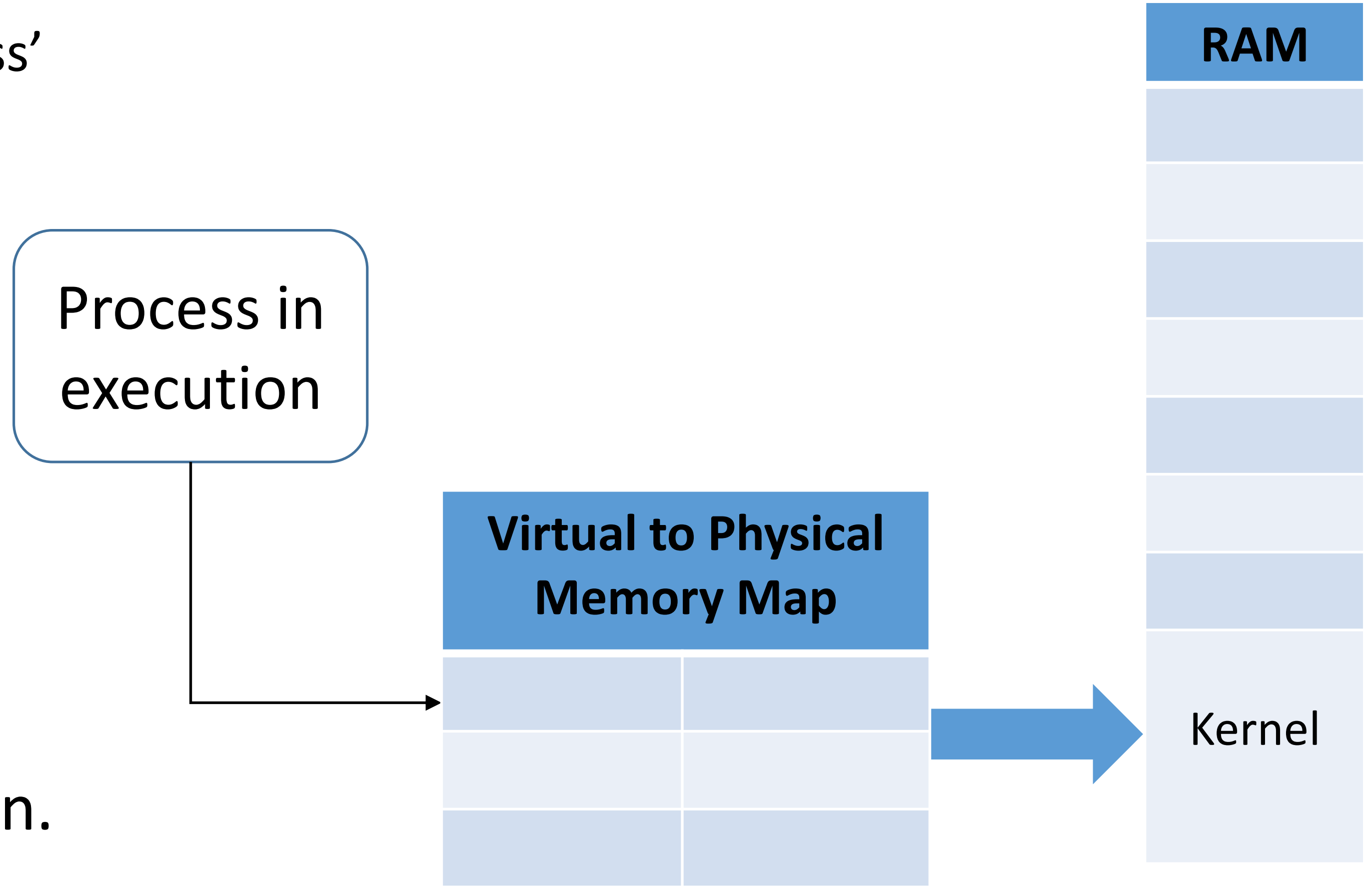
Inter Process Communication (IPC)
Process Synchronization
Deadlocks

CS3003D: Operating Systems

Need for IPC

- Each process has it's own virtual address space
  - Cannot view/access another process' address space
  - MMU maps the virtual address to Physical/RAM address
- Cannot guess/determine the physical address mapping



- How does one process communicate with another process?

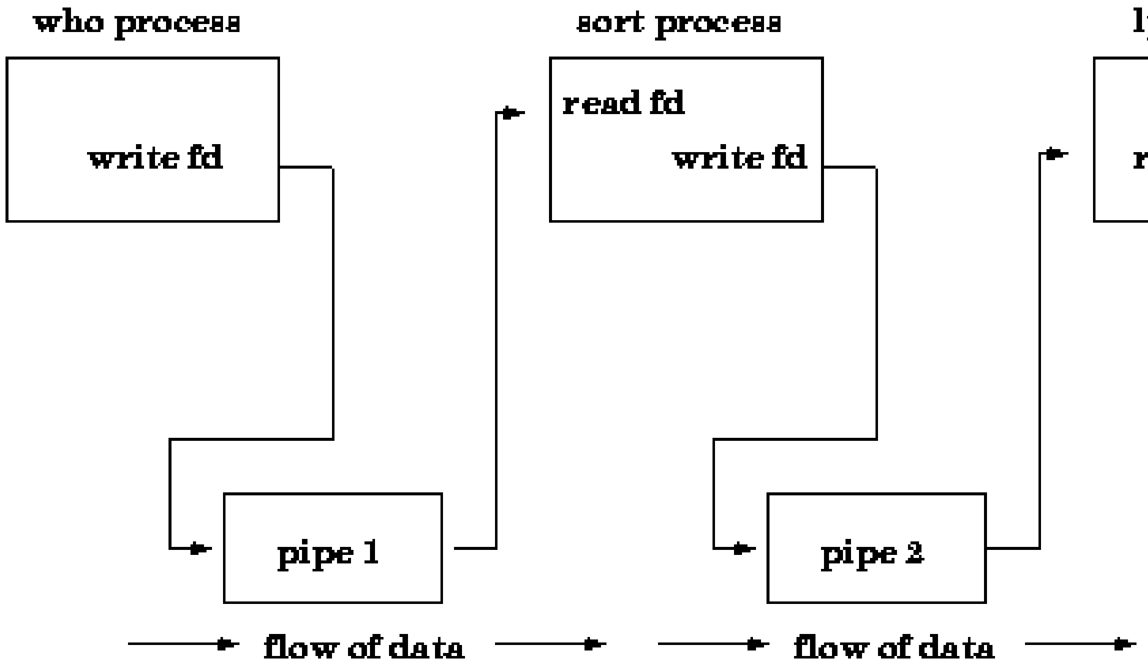
  Inter Process Communication.

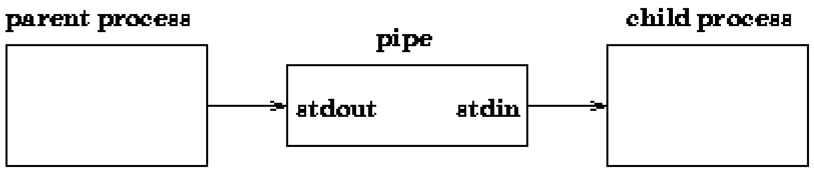
IPC

- Information sharing between processes
  - P1: data collection from the environment
  - P2: analyses the data collected
  - P3: actuates the external devices
- Convenient usage

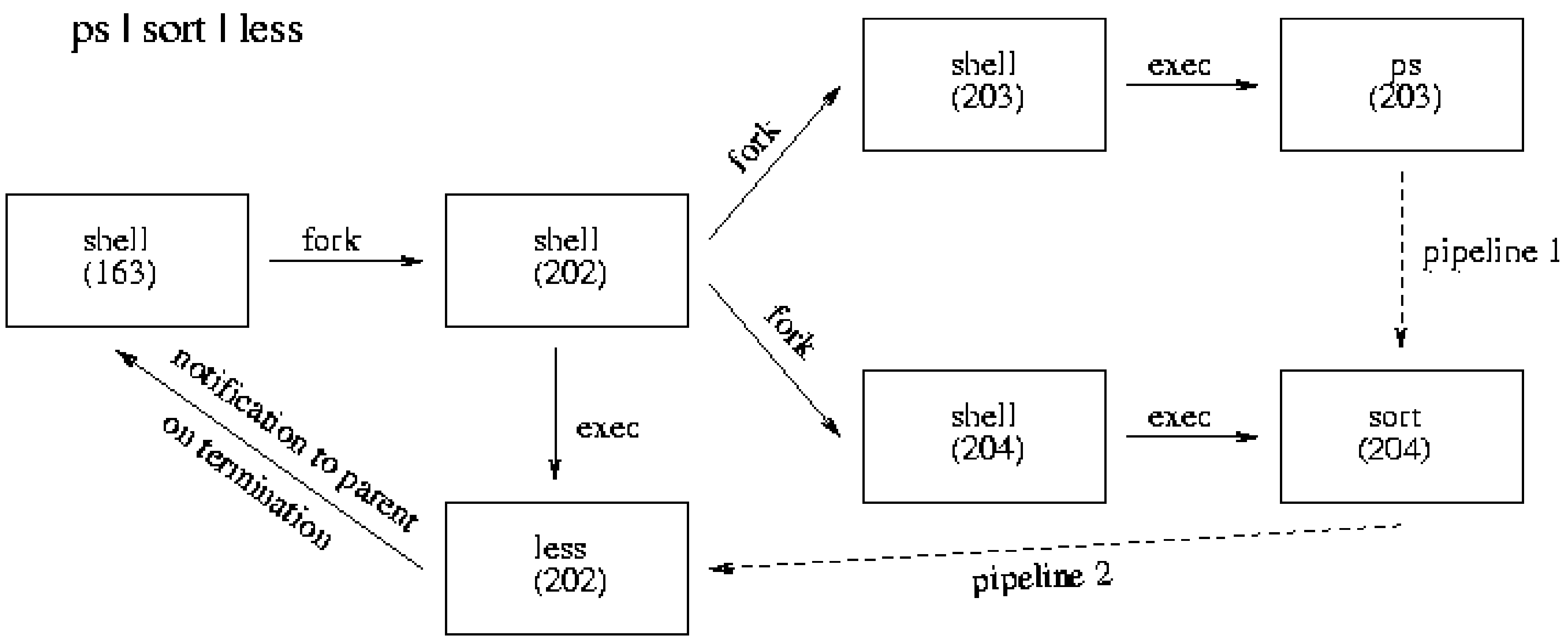
Pipes (unnamed pipes)

- who | sort | lpr





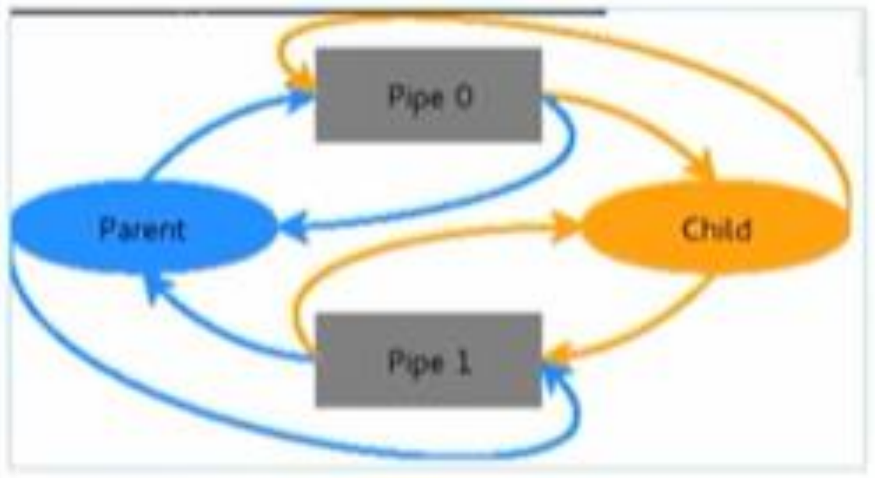
ps | sort | less

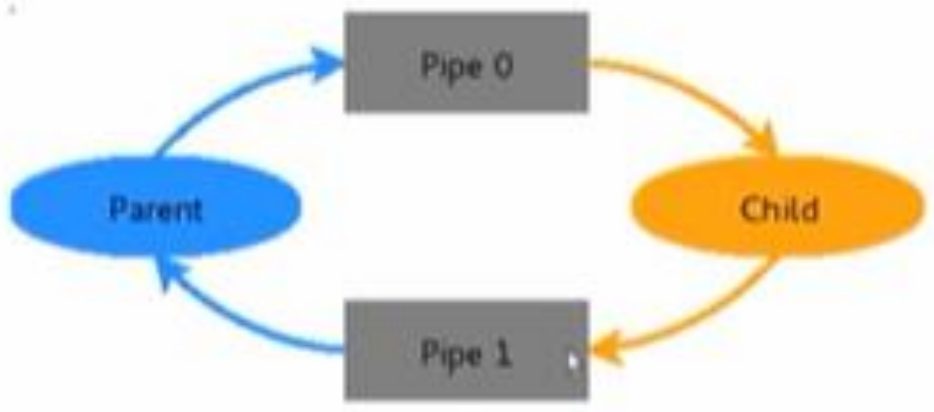

ps | sort | less
shell
(163)
fork
shell
(202)
fork
shell
(203)
exec
ps
(203)
pipeline 1
fork
shell
(204)
exec
sort
(204)
exec
less
(202)
pipeline 2
notification to parent
on termination

Pipes

- Always communication between child and parent processes
- Unidirectional
- fd[0] – reads from the pipe
- fd[1]- writes to a pipe

- Messages from parent to child
  - parent closes fd[0]
  - child closes fd[1]
- Messages from child to parent
  - parent closes fd[1]
  - child closes fd[0]





A pipe program

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

int main(void)
{
        int     fd[2], nbytes;
        pid_t   childpid;
        char    string[] = "Hello, world!\n";
        char    readbuffer[80];

        pipe(fd);

        if((childpid = fork()) == -1)
        {
                perror("fork");
                exit(1);
        }

        if(childpid == 0)
        {
                /* Child process closes up input side of pipe */
                close(fd[0]);

                /* Send "string" through the output side of pipe */
                write(fd[1], string, (strlen(string)+1));
                exit(0);
        }
        else
        {
                /* Parent process closes up output side of pipe */
                close(fd[1]);

                /* Read in a string from the pipe */
                nbytes = read(fd[0], readbuffer, sizeof(readbuffer));
                printf("Received string: %s", readbuffer);
        }

        return(0);
}
```

Named Pipe

- Also known as FIFO
- A named pipe can last as long as the system is up, beyond the life of the process
  - It can be deleted if no longer used
- Usually named pipe appears as a file; two or more processes can communicate by reading/writing from/to the file
- The named pipe resides in the kernel and not on a physical file system
- Function call:

```
int mkfifo(const char *pathname, mode_t mode);
mknod()
```

Named Pipe ... Contd.

Process A

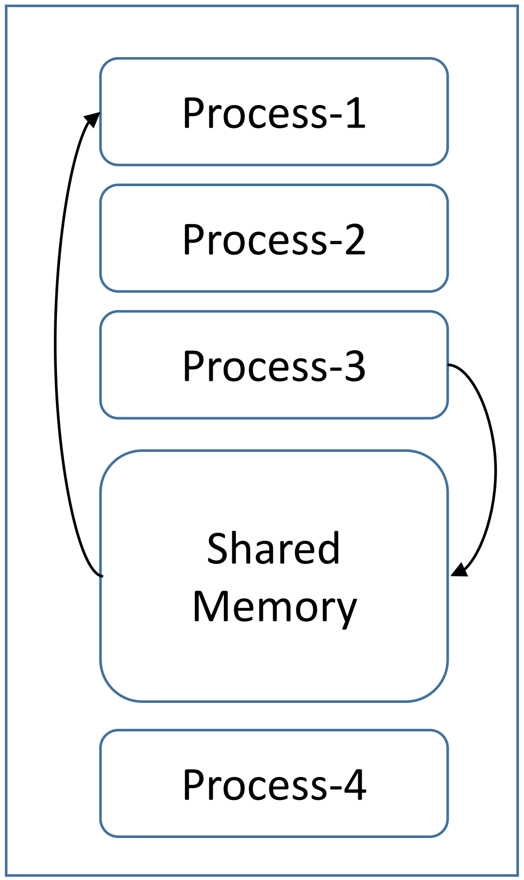
```
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <unistd.h>

#include <linux/stat.h>

#define FIFO_FILE       "MYFIFO"

int main(void)
{
        FILE *fp;
        char readbuf[80];

        /* Create the FIFO if it does not exist */
        umask(0);
        mknod(FIFO_FILE, S_IFIFO|0666, 0);

        while(1)
        {
                fp = fopen(FIFO_FILE, "r");
                fgets(readbuf, 80, fp);
                printf("Received string: %s\n", readbuf);
                fclose(fp);
        }

        return(0);
}
```

Process B

```
#include <stdio.h>
#include <stdlib.h>

#define FIFO_FILE       "MYFIFO"

int main(int argc, char *argv[])
{
        FILE *fp;

        if ( argc != 2 ) {
                printf("USAGE: fifoclient [string]\n");
                exit(1);
        }

        if((fp = fopen(FIFO_FILE, "w")) == NULL) {
                perror("fopen");
                exit(1);
        }

        fputs(argv[1], fp);

        fclose(fp);
        return(0);
}
```

Shared Memory

- Process creates an area in RAM so that other processes can access
- Reading from and writing to the shared memory space
- There is no intermediary such as message queues, pipes, etc.
- Adv.: Fast
- Prone to error; processes should synchronize



Shared Memory … Contd.

- shmget() is used to obtain access to a shared memory segment.
- int shmget(key_t key, size_t size, int shmflg);
  - key : unique key/ID
  - size : size in bytes of the requested shared memory
  - shmflg : initial access permissions and creation control flags
  - returns ID of the segment: shmID
- shmat() and shmdt() are used to attach and detach shared memory segments
  - Attach a process (address space of the process) to the shared memory, shmID
  - Detach shared memory

Shared Memory – Process01

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>

#define SHMSZ     27 // size of shared memory

main()
{
    char c;
    int shmid;
    key_t key;
    char *shm, *s;

    /* We'll name our shared memory segment "5678". */
    key = 5678;

    /* Create the segment. */
    if ((shmid = shmget(key, SHMSZ, IPC_CREAT | 0666)) < 0) {
        perror("shmget");
        exit(1);
    }

    /* Now attach the segment to the data space. */
    if ((shm = shmat(shmid, NULL, 0)) == (char *) -1) {
        perror("shmat");
        exit(1);
    }

    /* Now put some things into the memory for the other process to read. */
    s = shm;
    for (c = 'a'; c <= 'z'; c++)
        *s++ = c;
    *s = NULL;

    /*
     * Finally, we wait until the other process
     * changes the first character of our memory to '*',
     * indicating that it has read the memory area       */
    while (*shm != '*')
        sleep(1);

    exit(0);
}
```

Source: https://users.cs.cf.ac.uk/Dave.Marshall/C/node27.html

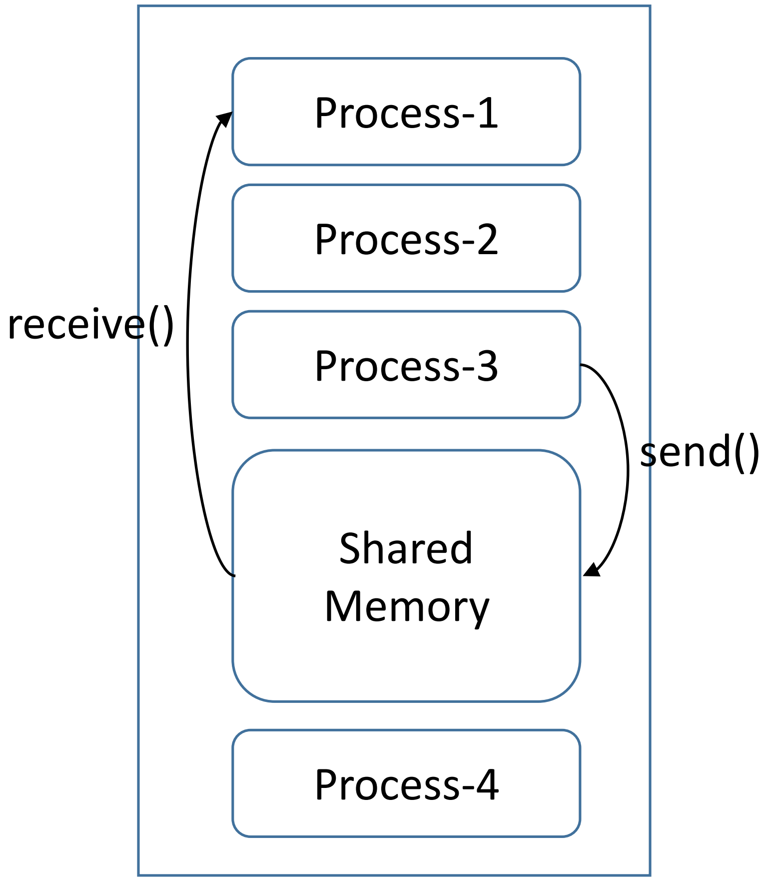
Shared Memory – Process02

```c
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>

#define SHMSZ     27

main()
{
    int shmid;
    key_t key;
    char *shm, *s;

    /* get the segment named "5678", created by the server. */
    key = 5678;

    /* Locate the segment. */
    if ((shmid = shmget(key, SHMSZ, 0666)) < 0) {
        perror("shmget");
        exit(1);
    }

    /* Attach the segment to the data space. */
    if ((shm = shmat(shmid, NULL, 0)) == (char *) -1) {
        perror("shmat");
        exit(1);
    }

    /* Now read what the server put in the memory  */
    for (s = shm; *s != NULL; s++)
        putchar(*s);
    putchar('\n');

    /*
     * Finally, change the first character of the
     * segment to '*', indicating we have read
     * the segment.
     */
    *shm = '*';

    exit(0);
}
```

Source: https://users.cs.cf.ac.uk/Dave.Marshall/C/node27.html

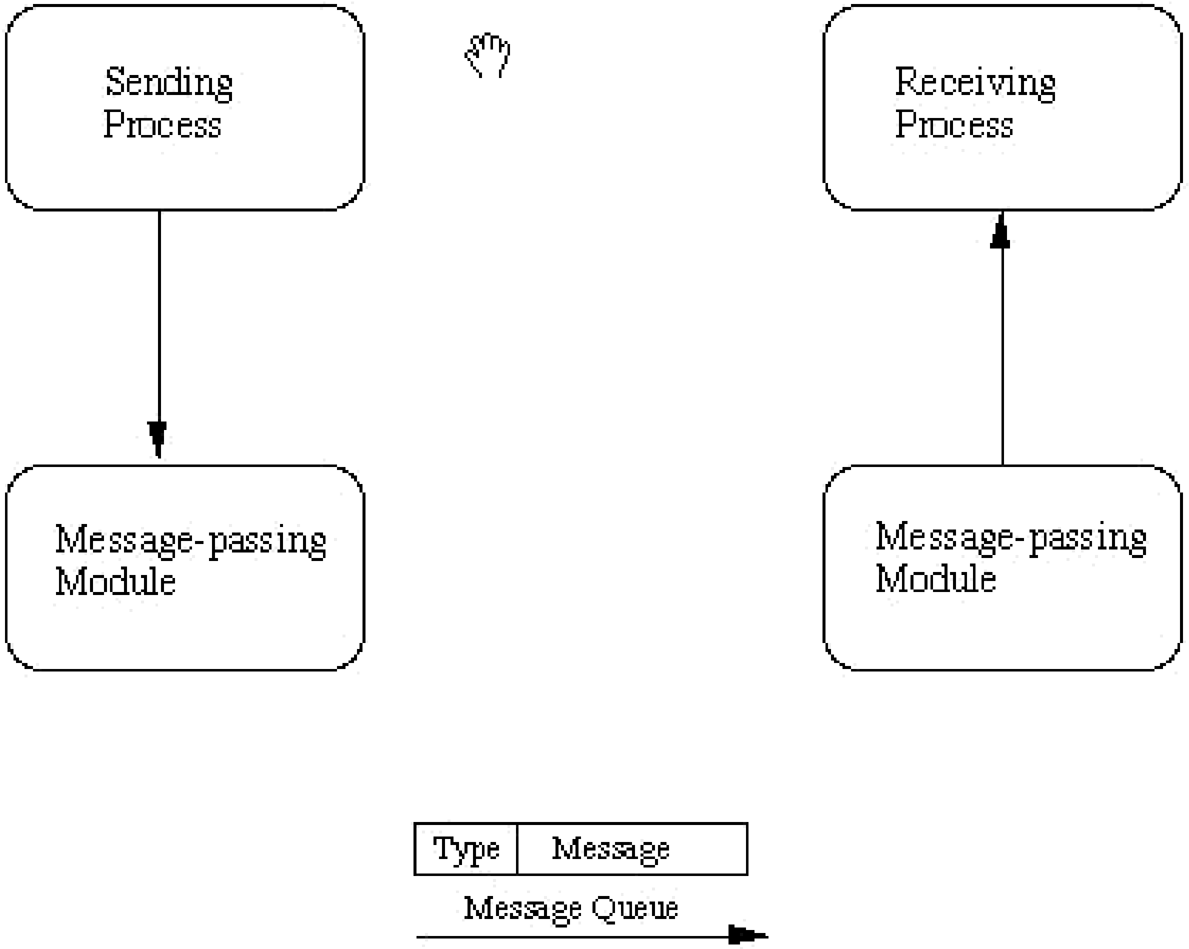
Message Passing

- Shared memory – reading and writing
- Message passing – send and receive
- System calls send() and receive() are used for communication
- Less error prone
- Slow because of system calls



Message Queues

- Easy implementation when communication between two or more processes



- int msgget(key_t key, int msgflg)

  creates or allocates a message queue

- int msgsnd(int msgid, const void *msgp, size_t msgsz, int msgflg)

  system call sends/appends a message into the message queue

- int msgrcv(int msgid, const void *msgp, size_t msgsz, long msgtype, int msgflg)

  system call retrieves the message from the message queue

Ref.: https://users.cs.cf.ac.uk/Dave.Marshall/C/node25.html

```c
// C Program for Message Queue (Writer Process)
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/msg.h>

// structure for message queue
struct mesg_buffer {
    long mesg_type;
    char mesg_text[100];
} message;

int main()
{
    key_t key;
    int msgid;

    // ftok to generate unique key
    key = ftok("progfile", 65);

    // msgget creates a message queue and returns identifier
    msgid = msgget(key, 0666 | IPC_CREAT);
    message.mesg_type = 1;

    printf("Write Data: ");
    gets(message.mesg_text);

    // msgsnd to send message
    msgsnd(msgid, &message, sizeof(message), 0);

    // display the message
    printf("Data send is : %s \n", message.mesg_text);

    return 0;
}
```

```c
// C Program for Message Queue (Reader Process)
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/msg.h>

// structure for message queue
struct mesg_buffer {
    long mesg_type;
    char mesg_text[100];
} message;

int main()
{
    key_t key;
    int msgid;

    // ftok to generate unique key
    key = ftok("progfile", 65);

    // msgget creates a message queue and returns identifier
    msgid = msgget(key, 0666 | IPC_CREAT);

    // msgrcv to receive message
    msgrcv(msgid, &message, sizeof(message), 1, 0);

    // display the message
    printf("Data Received is : %s \n",
                    message.mesg_text);

    // to destroy the message queue
    msgctl(msgid, IPC_RMID, NULL);

    return 0;
}
```

Synchronization

shared variable
int flag = 5

ProgramA

flag++

1) reg1 = flag
2) reg1 = reg1 + 1
3) flag = reg1

Output value of flag can be 5, 4, or 6 based on the way the processes are executing, when context switching happens

ProgramB

flag --

4) reg2 = flag
5) reg2 = reg2 - 1
6) flag = reg2

Scenario1

ProcessA
1)
2)
3) flag = 6
Context Switch
ProcessB
4)
5)
6) flag = 5

Scenario2

ProcessB
1)
2)
3) flag = 4
Context Switch
ProcessA
4)
5)
6) flag = 5

Scenario3

ProcessA
1) reg1=5
Context Switch
ProcessB
2) reg2 = 5
3) reg2 = 4
4) flag = 4
Context Switch
ProcessA
5) reg1 = 6
6) flag = 6

Scenario4

ProcessB
1) reg2=5
Context Switch
ProcessA
2) reg2 = 5
3) reg2 = 6
4) flag = 6
Context Switch
ProcessB
5) reg1 = 4
6) flag = 4

Race condition

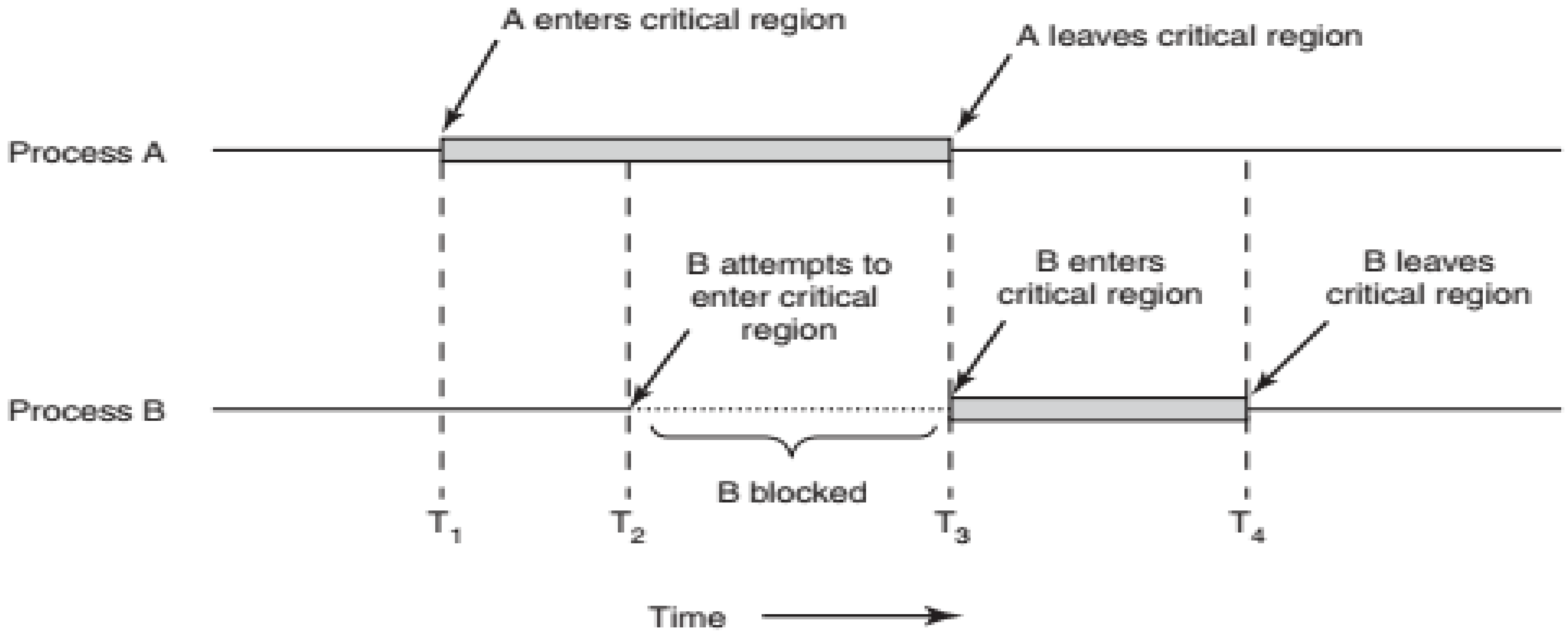
- Many processes manipulate the same data portion
- During concurrent execution outcome depends upon the order in which the access happens
- Incorrect data leads to misleading output
- Can be prevented by synchronization between processes

How to avoid race condition?

- Prohibit more than one process from reading and writing the shared data (critical section) at the same time

Three requirements for critical section problem

- Mutual Exclusion
  - No two processes may be simultaneously inside their critical regions
- Progress
  - No process running outside its critical region may block any process
- No starvation (bounded waiting)
  - No process should have to wait forever to enter its critical region



Solutions to critical section

- Disable interrupts
    - Context switches will not happen
    - Codes that execute in the kernel can only disable interrupts
    - User processes/application programs cannot disable interrupts

```
While(TRUE) {
   // code area
   disable_ interrupts()    < LOCK
   critical_section
   enable_interrupts()     < UNLOCK
   // other code area
}
```

Busy waiting

Shared
int turn = 1;

Process-1

```
while (TRUE) {
  while (turn == 2);      // LOCK
  critical_section
  turn = 2;               // UNLOCK
}
```

Process-2

```
while (TRUE) {
  while (turn == 1);      // LOCK
  critical_section
  turn = 1;               // UNLOCK
}
```

- Mutual exclusion achieved
- Busy waiting – resource wastage
  - When Process-2 executes first, always in loop; always in primary memory – either at READY state or at RUNNING state
- Progress condition is violated
  - Process-1 -> Process-2 -> Process-1 -> Process-2 ->

No Mutual Exclusion

Shared
p1_inside = false, p2_inside = false

Process-1

```
while (TRUE) {
  while (p2_inside == TRUE);      // LOCK
  p1_inside = TRUE;
  critical_section
  p1_inside = FALSE;              // UNLOCK
}
```

Process-2

```
while (TRUE) {
  while (p1_inside == TRUE);      // LOCK
  p2_inside = TRUE;
  critical_section
  p2_inside = FALSE;              // UNLOCK
}
```

```
while(p2_inside == TRUE);
// Context Switch (Process-2)
while(p1_inside == TRUE);
p2_inside = TRUE;
// Context Switch (Process-1)
p1_inside = TRUE;
```

- Mutual exclusion is not guaranteed
- Both processes can enter into critical section

Endless Wait

Shared
p1_inside = false, p2_inside = false

Process-1

```
while (TRUE) {
  p1_inside = TRUE;
  while (p2_inside == TRUE);      // LOCK
  critical_section
  p1_inside = FALSE;              // UNLOCK
}
```

Process-2

```
while (TRUE) {
  p2_inside = TRUE;
  while (p1_inside == TRUE);      // LOCK
  critical_section
  p2_inside = FALSE;              // UNLOCK
}
```

```
p1_inside = TRUE
// Context Switch (Process-2)
p2_inside = TRUE;
```

- Achieves Mutual exclusion
- Can it progress?
  - DEADLOCK!

Peterson's Solution

Shared
p1_inside, p2_inside, favoured

Process-1

```
while (TRUE) {
  p1_inside = TRUE;
  favoured = 2;
  while (p2_inside == TRUE AND favoured =2);  // LOCK
  critical_section
  p1_inside = FALSE;                // UNLOCK
}
```

Process-2

```
while (TRUE) {
  p2_inside = TRUE;
  favoured = 1;
  while (p1_inside == TRUE AND favoured =1);    // LOCK
  critical_section
  p2_inside = FALSE;                // UNLOCK
}
```

Breaking the deadlock as one of the processes is favoured

Bakery Algorithm – synchronization between N processes (N > 2)

- Proposed by Leslie Lamport
- Similar to token system in the bakeries/banks
    - Customers upon entering the bank is issued with the token
    - Waits until his/her turn arrives
    - Dispense the token and the service is rendered

Ref.: http://www.cs.umd.edu/~shankar/412-S99/note-7.html

Simplified version of Bakery algorithm

- Each process is numbered 0 to N-1
- Each process i has an integer variable num[i], initially 0, that is readable by all processes but writeable by process i only

```
Entry(i) {
  num[i] = MAX( num[0], num[1], ... , num[N-1] ) + 1 ;
  for p = 0 to N-1 do  {
      while (num[p] != 0 AND num[p] < num[i])   do no-op ;
  }
}
```

Lock

Critical section

```
Exit(i) {
  num[i] = 0 ;
}
```

Unlock

Example

num[i]	P0	P1	P2	P3	P4
Initial	0	0	0	0	0
P2	0	0	1	0	0
P3	0	0	0	2	0
P4	0	0	0	0	3
P0	4	0	0	0	0
P1	4	5	0	0	0
Final	4	5	1	2	3

num[i] = MAX(num[0], num[1], ... , num[N-1]) + 1

num[i]	P0	P1	P2	P3	P4
Initial	4	5	1	2	3
P2	4	5	0	2	3
P3	4	5	0	0	3
P4	4	5	0	0	0
P0	0	5	0	0	0
P1	0	0	0	0	0
Final	0	0	0	0	0

```
for p = 0 to N-1 do  {
        while(num[p] != 0 AND num[p] < num[i]) do no-op ;
  }
```

Problem!

Assumption: No two processes get the same token

When two process gets the same num[i] value (same token)

Two processes enter into the critical section

Bakery algorithm

MAX() is no more assumed to be atomic

Introduction of an array of N booleans, choosing[i] = FALSE

```
Entry(i) {
  choosing[i] = TRUE;
  num[i] = MAX( num[0], num[1], ... , num[N-1] ) + 1 ;
  choosing[i] = FALSE;
  for p = 0 to N-1 do  {
      while(choosing[p]);
      while (num[p] != 0 AND (num[p],p) < (num[i],i))   do no-op ;
  }
}                       (a,b) < (c,d) is equivalent to (a<c) or ((a==c) AND (b<d))
```

Lock

Critical section

```
Exit(i) {
  num[i] = 0 ;
}
```

Unlock

Example

num[i]	P0	P1	P2	P3	P4
Initial	0	0	0	0	0
P2	0	0	1	0	0
P3	0	0	0	2	0
P4	0	0	0	0	2
P0	3	0	0	0	0
P1	3	4	0	0	0
Final	3	4	1	2	2

num[i]	P0	P1	P2	P3	P4
Initial	3	4	1	2	2
P2	3	4	0	2	2
P3	3	4	0	0	2
P4	3	4	0	0	0
P0	0	4	0	0	0
P1	0	0	0	0	0
Final	0	0	0	0	0

num[i] = MAX(num[0], num[1], ... , num[N-1]) + 1

```
for p = 0 to N-1 do  {
      while(num[p] != 0 AND num[p] < num[i]) do no-op ;
  }
```

In the case of a tie in num[i],

(num[p],p) < (num[i],i) ensures that that the process with lesser ID prevails

Hardware Locks

lock = 0

Process-1
```
while(1) {
   while(lock != 0);
    lock = 1;          // lock
    // critical_section
   lock = 0;           // unlock
}
```

Process-2
```
while(1) {
   while(lock != 0);
    lock = 1;          // lock
    // critical_section
   lock = 0;           // unlock
}
```

lock = 0

Process-1

while(lock != 0)

Context Switch

Process-2

while(lock !=0)

lock =1

Context Switch

Process-1

lock = 1

Both processes in critical section

Mutual exclusion not possible

```
while(lock != 0);     /* should be made
lock = 1;                atomic */
```

Test and Set


```
int test_and_set(int *L) {
  int prev = *L;
  *L = 1;
  return prev;
}

while(1) {
 while (test_and_set(&lock) == 1);
  // critical_section
  lock = 0;
}
```

- An atomic function
- Only one process can access the test_and_set function
- Hardware ensures only one process can execute the TAS at a time before another process wants to execute

- test_and_set will read lock=0 and set set lock=1
- Other processes will see lock=0 and infinitely be in the while loop.

xchg instruction … Intel equivalent for test and set

```
int xchg(int *L, int val) {
  int prev = *L;
  *L = val;
  return prev;
}
```

- It is an atomic instruction

```
int xchg(addr, value){
  %eax = value
  xchg %eax, (addr)
}
```

```
void acquire(int *locked) {
  while(1){
    if(xchg(locked, 1) == 0)
      break;
  }
}
```

```
void release(*locked) {
  locked = 0;
}
```

Spinlock

Process1
Acquire(&locked)
Critical_section
Release(&locked)

Process2
Acquire(&locked)
Critical_section
Release(&locked)

- Only one process can acquire the lock
- Meanwhile other processes wait in a loop for the lock
- When a process releases the lock, it becomes available for other processes

Mutex

- Spinlock is good at short critical sections such as counter increment, accessing array element, etc.
- When period of wait is longer, spinlock is not that effective

- Alternative: Mutex

Mutex

- sleep() – running state to block state
- wakeup() – block state to ready queue

```
int xchg(addr, value){
  %eax = value
  xchg %eax, (addr)
}
```

```
void lock(int *locked) {
  while(1){
    if(xchg(locked, 1) == 0)
      break;
    else
      sleep()
  }
}
```

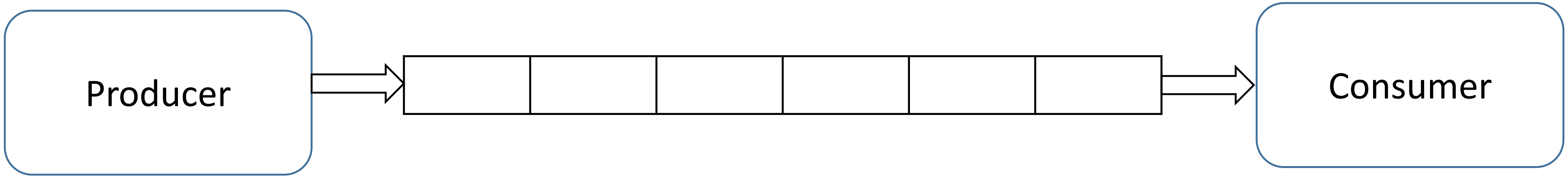
```
void unlock(*locked) {
  locked = 0;
  wakeup();
}
```

Thundering herd problem

- when a large number of processes (or threads) waiting for an event are woken up when that event occurs
  - but only one process is able to handle the event
- When the processes wake up, they will each try to handle the event, but only one will win
- All waiting processes wakeup simultaneously
- Large number of context switches
- Can lead to starvation

```
void lock(int *locked) {
  while(1){
    if(xchg(locked, 1) == 0)
      break;
    else
       add process to queue
      sleep()
  }
}

void unlock(*locked) {
  locked = 0;
  remove process P from queue
  wakeup(P);
}
```

Semaphore - another synchronization primitive



Producer – Consumer problem
(Bounded buffer problem)

- Problem
  - Producer produces fast, but buffer is FULL
  - Consumer consumes fast, but buffer is EMPTY

mutex, full, and empty are mutexes

Producer

```
while(TRUE) {
  item = produce_item;
  if (count == N) sleep(empty);
  lock(mutex);
  insert_item_to_buffer;
  count++;
  unlock(mutex);
  if(count == 1) wakeup(full);
}
```

Consumer

```
while(TRUE) {
  if(count == 0) sleep(full);
  lock(mutex);
  item = remove_item_from_buffer;
  count--;
  unlock(mutex);
  if(count == N-1) wakeup(empty);
  item = consume_item;
}
```

Problem arises when

Count = 0

Context switch – Producer

```
item = produce_item;
if (count == N) sleep(empty);
lock(mutex);
insert_item_buffer;
count++;
unlock(mutex);
if(count == 1) wakeup(full);
```

Context switch – Consumer

```
if(count == 0) sleep(full);
```

- Consumer uses the old value of count [=0]
  - Consumer waits
- Producer goes on producing
  - Until the buffer is full
- Further, both producer and consumer wait infinitely

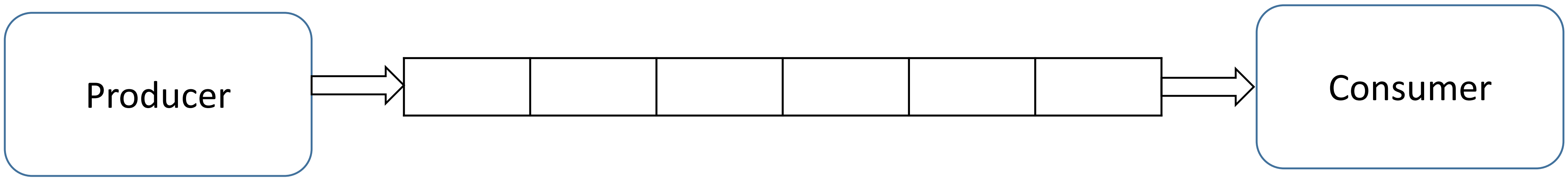
- Three mutexes – *full, empty, and mutex* – are insufficient to solve the problem

Semaphore – proposed by Dijkstra (1965)

- Two atomic functions – down() and up(); also called P() and V()
- S is a shared memory location

```
void down(int *S) {
  while(*S <= 0);
    *S--;
}
```

```
void up(int *S)  {
  *S++;
}
```



- Two semaphores – empty and full
- Indicate the number of empty and full slots in the buffer respectively

full and empty are semaphores

Producer

```
while(TRUE) {
  item = produce_item;
  down(empty);
  insert_item_to_buffer;
  up(full)
}
```

Consumer

```
while(TRUE) {
  down(full);
  remove_item_from_buffer;
  up(empty);
  item = consume_item;
}
```

full and empty are semaphores initially buffer is FULL

Producer

```
while(TRUE) {
  item = produce_item;
  down(empty);
  insert_item_to_buffer;
  up(full)
}
```

Consumer

```
while(TRUE) {
  down(full);
  remove_item_from_buffer;
  up(empty);
  item = consume_item;
}
```

full and empty are semaphores
initially buffer is EMPTY

Producer

```
while(TRUE) {
  item = produce_item;
  down(empty);
  insert_item_to_buffer;
  up(full)
}
```

Consumer

```
while(TRUE) {
  down(full);
  remove_item_from_buffer;
  up(empty);
  item = consume_item;
}
```

Synchronized access – using mutex
full and empty are semaphores

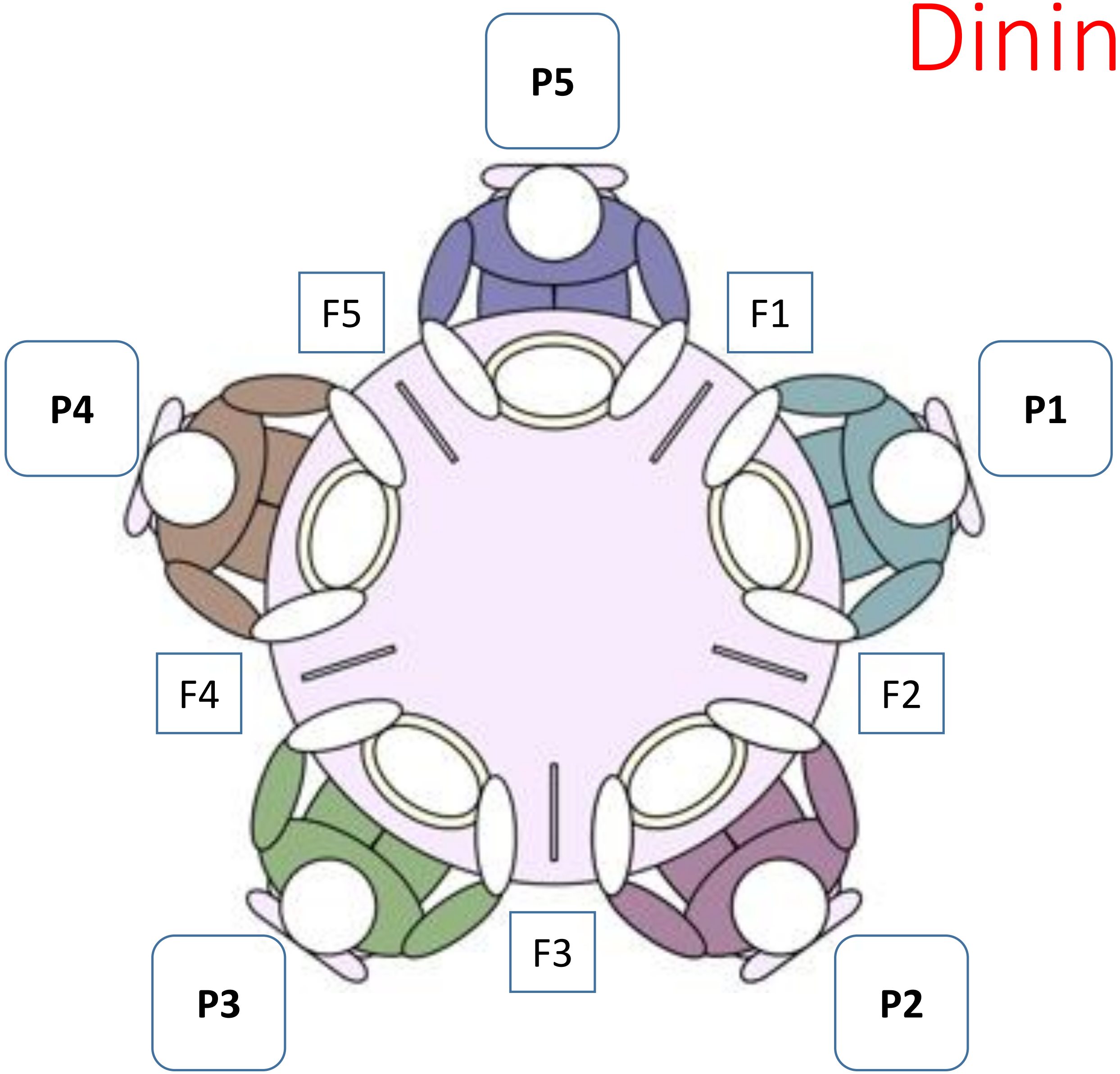
Producer

```
while(TRUE) {
  item = produce_item;
  down(empty);
  lock(mutex);
  insert_item_to_buffer;
  unlock(mutex);
  up(full)
}
```

Consumer

```
while(TRUE) {
  down(full);
  lock(mutex);
  remove_item_from_buffer;
  unlock(mutex);
  up(empty);
  item = consume_item;
}
```

Dining Philosopher problem



- Philosophers are independent
- At any instant, a philosopher is either eating or thinking
- When a philosopher wants to eat, he uses two forks - one from the left and one from the right
- When a philosopher wants to think, he keeps down both forks at their original place.
- Problem: None of the philosophers should STARVE

Fig. Source: https://medium.com/@zinayouhan33/dining-philosopher-problem-and-solution-9a273a5fa614

Possible Solution - 1

```
philosopher(int i) {
  while(TRUE) {
    think();
    pick_fork(R);
    pick_fork(L);
    eat();
    put_fork(L);
    put_fork(R);
  }
}
```

- What happens if only Philosophers P2 and P4 are given priority?
  - P2 uses F2 and F3
  - P4 uses F4 and F5
  - P1, P3, and P5 never get the forks and are starved
- What happens is all philosophers think and eat at the same time with think_time_quanta the same?
  - All philosophers pick their right forks
  - None gets the left fork
  - Deadlock situation: indefinite execution without any progress
- Solution is not ideal

Possible Solution - 2

```
philosopher(int i) {
   while(TRUE) {
      think();
      pick_fork(R);
      if (available(L)) {
            pick_fork(L);
            eat();
            put_fork(L);
            put_fork(R);
    }
    else {
            put_fork(R)
            sleep(T);
   }
}
```

- Situation when all the philosophers execute simultaneously; think and eat at the same time
- Again leads to deadlock as all the philosophers lift the right fork, upon finding the unavailability of the left fork, put back the right fork
- Solution is not ideal
- What if the philosophers sleep for a random_time rather than fixed_time T?
    - Still starvation cannot be ruled out

Possible Solution – 3 (using mutex)

```
philosopher(int i) {
  while(TRUE) {
    think();
    lock(mutex);
    pick_fork(R);
    pick_fork(L);
    eat();
    put_fork(L);
    put_fork(R);
    unlock(mutex);
  }
}
```

- Protection of critical section
- Prevents deadlock
- But the problem/issue is only one philosopher can eat at any given time
  - Since the mutex is shared by all the philosophers

Possible Solution – 3 (using semaphore)

- Uses N semaphores (s[1], s[2], … s[N]) all initialized to zero
- Philosopher can be in three states; HUNGRY, EATING, and THINKING
- A philosopher can EAT only if the neighbours do not eat

```
void philosopher(int i) {
  while(TRUE) {
    think();
    pick_forks(i);
    eat();
    put_forks(i);
  }
}
```

```
void pick_forks(int i) {
  lock(mutex);
  state[i] = HUNGRY;
  test(i);
  unlock(mutex);
  down(s[i]);
}
```

```
void put_forks(int i) {
  lock(mutex);
  state[i] = THINKING;
  test(LEFT);
  test(RIGHT);
  unlock(mutex);
  }
```

```
void test(int i) {
  if(state[i] = HUNGRY && state[LEFT]!= EATING && state[RIGHT] != EATING) {
    state[i] = EATING;
    up(s[i]);
  }
}
```

	P1	P2	P3	P4	P5
State	T	T	T	T	T
Semaphore	0	0	0	0	0

	P1	P2	P3	P4	P5
State	T	H	T	T	T
Semaphore	0	1	0	0	0

	P1	P2	P3	P4	P5
State	T	E	T	T	T
Semaphore	0	0	0	0	0

	P1	P2	P3	P4	P5
State	T	E	H	T	T
Semaphore	0	0	0	0	0

P3 gets blocked

	P1	P2	P3	P4	P5
State	T	T	H	T	T
Semaphore	0	0	0	0	0

	P1	P2	P3	P4	P5
State	T	T	E	T	T
Semaphore	0	0	1	0	0

	P1	P2	P3	P4	P5
State	T	T	E	T	T
Semaphore	0	0	0	0	0

Deadlocks

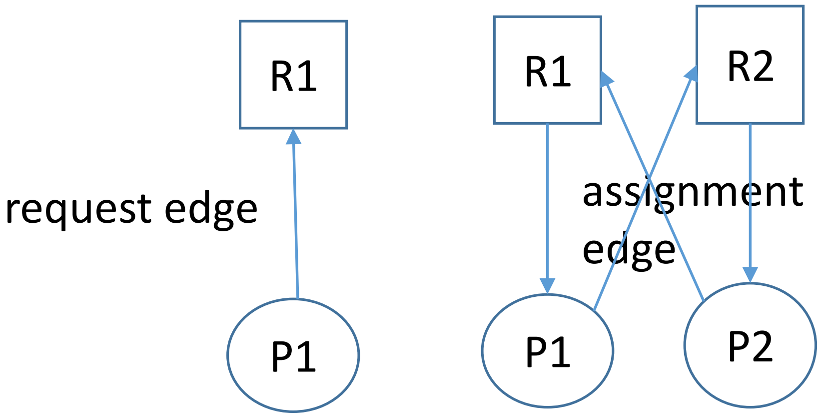
- System consists of resources
- Resource types $R_1, R_2, \ldots, R_m$

  CPU cycles, memory space, I/O devices
- Each resource type R_i has W_i instances.
- Each process utilizes a resource as follows:
  - request
  - use
  - release

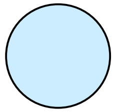
Ref.: https://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/7_Deadlocks.html

Resource-Allocation Graph

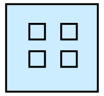
- A set of vertices *V* and a set of edges *E*.
- V is partitioned into two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system
- **request edge** – directed edge $P_i \rightarrow R_j$
- **assignment edge** – directed edge $R_j \rightarrow P_i$



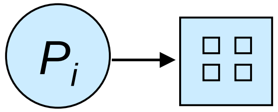
- Process



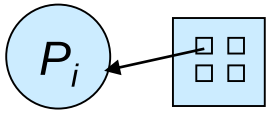
- Resource Type with 4 instances

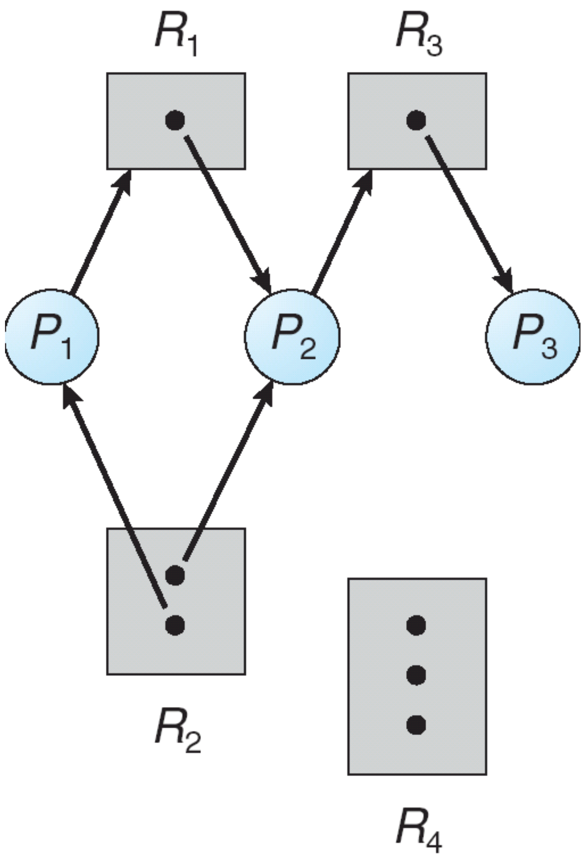


- P_i requests instance of R_j

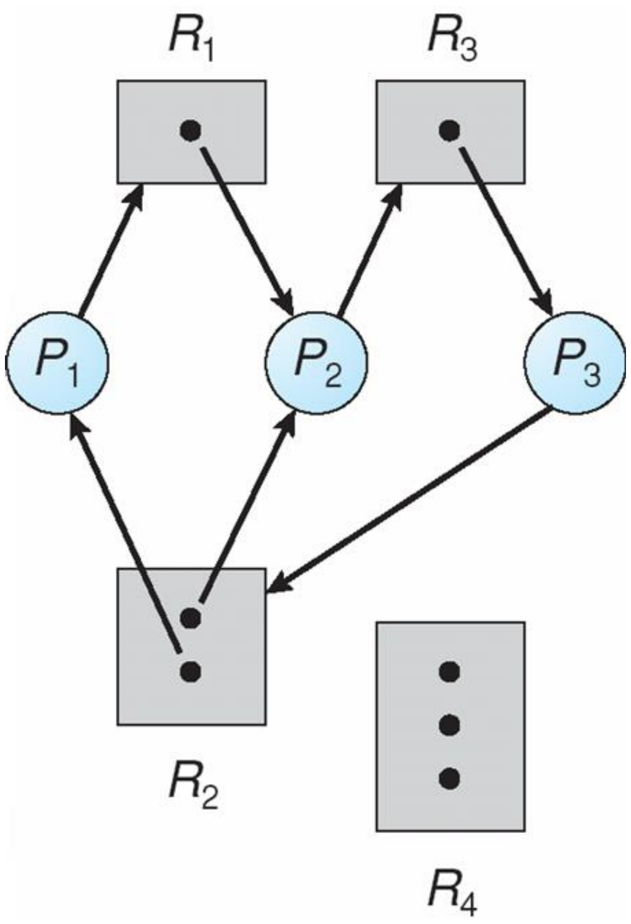


- P_i is holding an instance of R_j



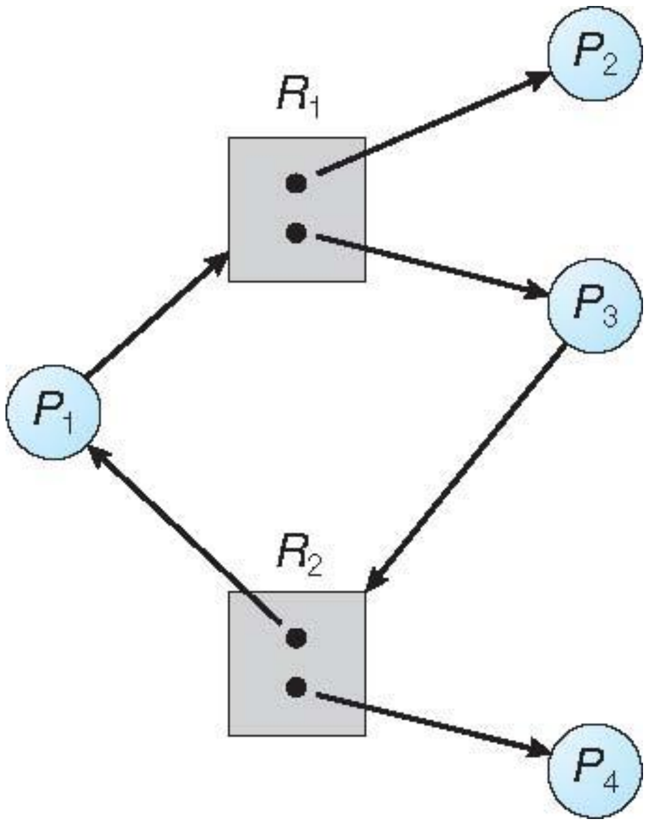


Resource Allocation Graph With a Deadlock



- Presence of a cycle in a graph is a potential deadlock situation
- Need not always end in a deadlock situation, but probably can end up

Graph With A Cycle But No Deadlock



- If graph contains no cycles $\Rightarrow$ no deadlock
- If graph contains a cycle $\Rightarrow$
  - if only one instance per resource type, then deadlock
  - if several instances per resource type, possibility of deadlock

Deadlock Characterization

- Deadlock can arise if the following four conditions hold simultaneously:
  - **Mutual exclusion**: only one process at a time can use a resource
  - **Hold and wait**: a process holding at least one resource is waiting to acquire additional resources held by other processes
  - **No preemption**: a resource can be released only voluntarily by the process holding it, after that process has completed its task
  - **Circular wait**: there exists a set $\{P_0, P_1, \ldots, P_n\}$ of waiting processes such that P_0 is waiting for a resource that is held by P_1, P_1 is waiting for a resource that is held by P_2, …, P_{n-1} is waiting for a resource that is held by P_n, and P_n is waiting for a resource that is held by P_0.

Handling deadlocks

- Ensure that the system will never enter a deadlock state:
  - Deadlock prevention
  - Deadlock avoidance
- Allow the system to enter a deadlock state and then recover

Deadlock Prevention

- **Mutual Exclusion** – not required for sharable resources (e.g., read-only files); must hold for non-sharable resources
- **Hold and Wait** – must guarantee that whenever a process requests a resource, it does not hold any other resources
  - Require process to request and be allocated all its resources before it begins execution, or allow process to request resources only when the process has none allocated to it.
  - Low resource utilization; starvation possible

Deadlock Prevention … Contd.

- **No Preemption** –
  - If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released
  - Preempted resources are added to the list of resources for which the process is waiting
- **Circular Wait** – impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration

Deadlock Avoidance

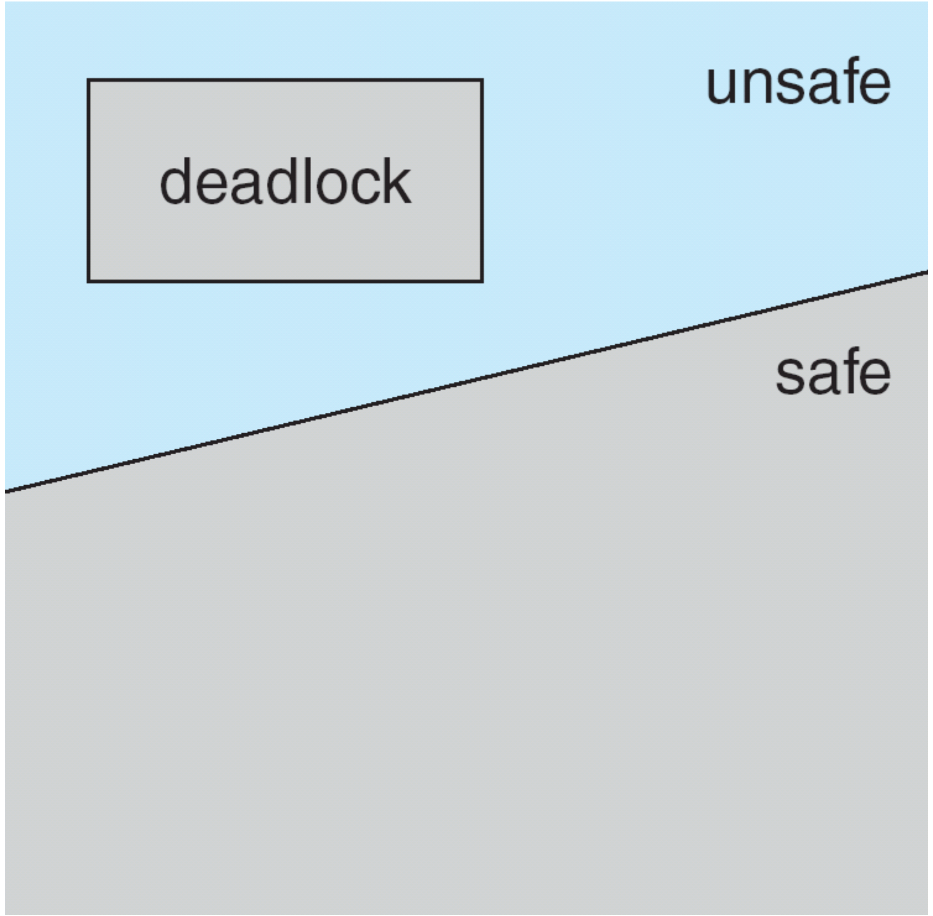
- Requires that the system has some additional a priori information available
  - Simplest and most useful model requires that each process declare the maximum number of resources of each type that it may need
  - The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition
  - Resource-allocation state is defined by the number of available and allocated resources, and the maximum demands of the processes

Safe State

- When a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state
- System is in safe state if there exists a sequence <P_1, P_2, …, P_n> of ALL the processes in the systems such that for each P_i, the resources that P_i can still request can be satisfied by currently available resources + resources held by all the P_j, with $j < I$
- That is:
  - If P_i resource needs are not immediately available, then P_i can wait until all P_j have finished
  - When P_j is finished, P_i can obtain needed resources, execute, return allocated resources, and terminate
  - When P_i terminates, P_{i+1} can obtain its needed resources, and so on

Safe State … Contd.

- If a system is in safe state $\Rightarrow$ no deadlocks
- If a system is in unsafe state $\Rightarrow$ possibility of deadlock
- Avoidance $\Rightarrow$ ensure that a system will never enter an unsafe state.



Deadlock Avoidance Algorithms

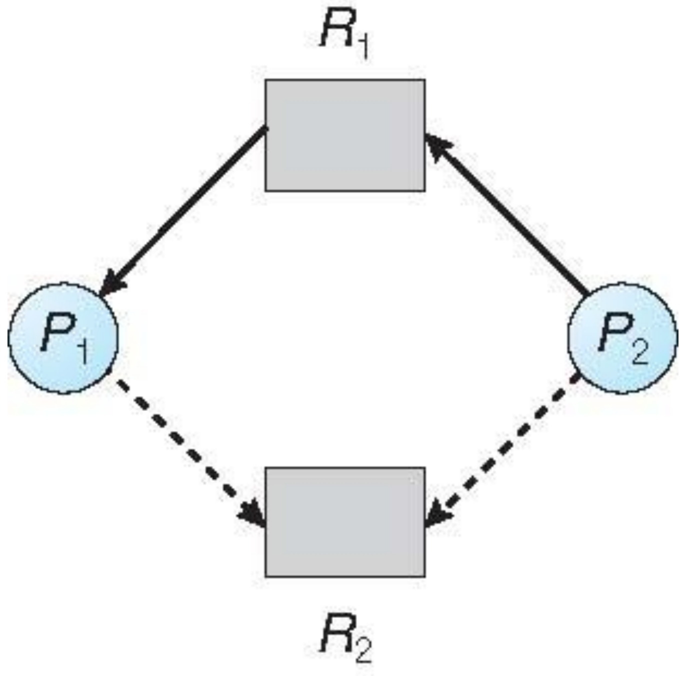
- Single instance of a resource type
  - Use a resource-allocation graph

- Multiple instances of a resource type
  - Use the Banker's algorithm

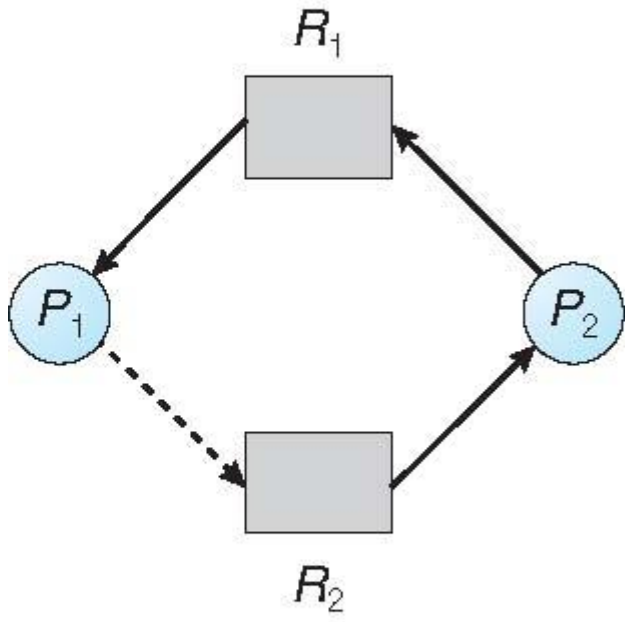
Resource-Allocation Graph Scheme

- Claim edge $P_i \rightarrow R_j$ indicated that process P_j may request resource R_j; represented by a dashed line
- Claim edge converts to request edge when a process requests a resource
- Request edge converted to an assignment edge when the resource is allocated to the process
- When a resource is released by a process, assignment edge reconverts to a claim edge
- Resources must be claimed *a priori* in the system

Claim edge



Unsafe state



Suppose that process P_i requests a resource R_j

The request can be granted only if **converting the request edge to an assignment edge does not result in the formation of a cycle** in the resource allocation graph

Banker’s Algorithm

- Multiple instances
- Each process must indicate a priori the maximum resource usage
- When a process requests a resource, it may have to wait
- When a process gets all its resources it must return them in a finite amount of time

Banker's Algorithm … Contd.

Let n = number of processes, and m = number of resources types

- **Available**: Vector of length m. If available $[j] = k$, there are k instances of resource type R_j available
- **Max**: n x m matrix. If $Max\,[i,j] = k$, then process P_i may request at most k instances of resource type R_j
- **Allocation**: n x m matrix. If Allocation$[i,j] = k$ then P_i is currently allocated k instances of R_j
- **Need**: n x m matrix. If $Need[i,j] = k$, then P_i may need k more instances of R_j to complete its task

$$Need\,[i,j] = Max[i,j] - Allocation\,[i,j]$$

Safety Algorithm

Let Work and Finish be vectors of length m and n, respectively. Initialize:

Work = Available

Finish [i] = false for i = 0, 1, …, n-1

1. Find an i such that both:

(a) Finish [i] = false

(b) $Need_i \leq Work$

If no such i exists, go to step 4

2. $Work = Work + Allocation_i$

Finish[i] = true

go to step 2

If Finish [i] == true for all i, then the system is in a safe state

Resource-Request Algorithm for Process P_i

$Request_i$ = request vector for process P_i. If $Request_i[j] = k$ then process P_i wants k instances of resource type R_j

1. If $Request_i \leq Need_i$ go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim
2. If $Request_i \leq Available$, go to step 3. Otherwise P_i must wait, since resources are not available
3. Pretend to allocate requested resources to P_i by modifying the state as follows:

$$Available = Available - Request_i$$
$$Allocation_i = Allocation_i + Request_i$$
$$Need_i = Need_i - Request_i$$

- If safe $\Rightarrow$ the resources are allocated to P_i
- If unsafe $\Rightarrow$ P_i must wait, and the old resource-allocation state is restored

Example

- 5 processes P_0 through P_4; 3 resource types: A (10 instances), B (5 instances), and C (7 instances)
- Snapshot at time T_0:

	Allocation	Max	Available	Need (Max-Allocation)
	A B C	A B C	A B C	A B C
P_0	0 1 0	7 5 3	3 3 2	7 4 3
P_1	2 0 0	3 2 2		1 2 2
P_2	3 0 2	9 0 2		6 0 0
P_3	2 1 1	2 2 2		0 1 1
P_4	0 0 2	4 3 3		4 3 1

Is the sequence < P_1, P_3, P_4, P_2, P_0> in a SAFE STATE?

Work = Avail = [3 3 2]   Finish[5] ={F, F, F, F, F}

P1
Need[1] <= Work & Finish[1] = F   [1 2 2] <= [3 3 2]
Work = Work + Alloc[1]   Work = [3 3 2] + [2 0 0] = [5 3 2]
Finish[1] = T;

P3
Need[3] <= Work & Finish[3] = F   [0 1 1] <= [5 3 2]
Work = Work + Alloc[3]   Work = [5 3 2] + [2 1 1] = [7 4 3]
Finish[3] = T

P4
Need[4] <= Work & Finish[4] = F   [4 3 1] <= [7 4 3]
Work = Work + Alloc[3]   Work = [7 4 3] + [0 0 2] = [7 4 5]
Finish[4] = T

P2
Need[2] <= Work & Finish[2] = F   [6 0 0] <= [7 4 5]
Work = Work + Alloc[1]   Work = [7 4 5] + [3 0 2] = [10 4
Finish[2] = T;

P0
Need[0] <= Work & Finish[0] = F   [7 4 3] <= [10 4 7]
Work = Work + Alloc[0]   Work = [5 3 2] + [2 1 1] = [7 4 3
Finish[0] = T

The sequence < P_1, P_3, P_4, P_2, P_0> is in a SAFE STATE

Can request for (1,0,2) by P1 be granted?

req[1] <= need[1]	[1 0 2] <= [3 3 2]
req[1] <= avail	[1 0 2] <= [3 3 2]
avail = avail - req[1]	avail = [3 3 2] – [1 0 2] = [2 3 0]
alloc[1] = alloc[1] + req[1]	alloc[1] = [2 0 0] + [1 0 2] = [3 0 2]
need[1] = need[1] - req[1]	need[1] = [1 2 2] – [1 0 2] = [0 2 0]

Safe state; request granted

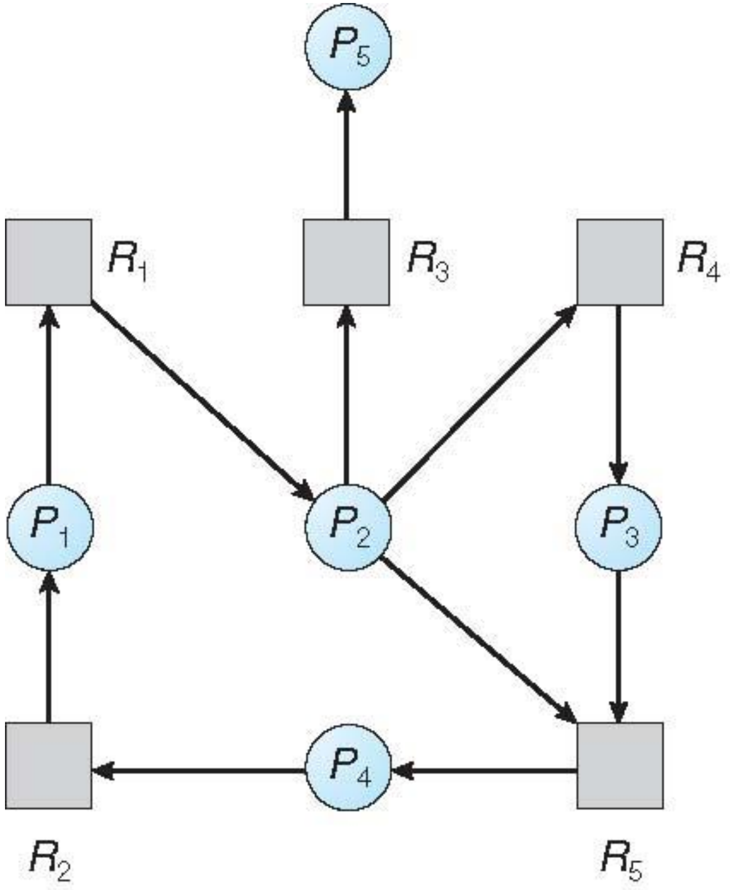
	Allocation	Need	Available
	A B C	A B C	A B C
P_0	0 1 0	7 4 3	2 3 0
P_1	3 0 2	0 2 0	
P_2	3 0 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

- Executing safety algorithm does the sequence < $\mathbf{P_1}$, $\mathbf{P_3}$, $\mathbf{P_4}$, $\mathbf{P_0}$, $\mathbf{P_2}$> satisfy safety requirement?
- Can request for (3,3,0) by $\mathbf{P_4}$ be granted?
- Can request for (0,2,0) by $\mathbf{P_0}$ be granted?

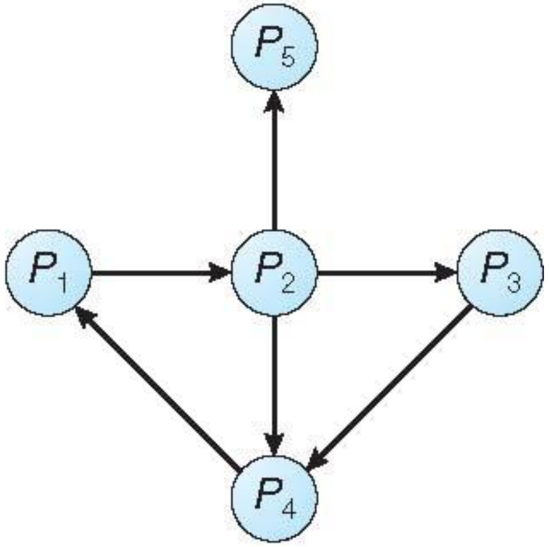
Deadlock Detection

- Single Instance of Each Resource Type
  - Maintain **wait-for** graph
    - Nodes are processes
    - $P_i \rightarrow P_j$ if P_i is waiting for P_j
  - Periodically invoke an algorithm that searches for a cycle in the graph. If there is a cycle, there exists a deadlock
  - An algorithm to detect a cycle in a graph requires an order of n^2 operations, where n is the number of vertices in the graph

Resource-Allocation Graph and Wait-for Graph



(a)



(b)

Detection algorithm

1. Let Work and Finish be vectors of length m and n, respectively
   (a) Work = Available
   (b) For i = 1,2, …, n, if $Allocation_i \neq 0$, then
       Finish[i] = false; otherwise, Finish[i] = true

2. Find an index i such that both:
   (a) Finish[i] == false
   (b) $Request_i \leq Work$

   If no such i exists, go to step 4

3. $Work = Work + Allocation_i$ and set Finish[i] = true
   Go to step 2

4. If Finish[i] == false, for some i, $1 \leq i \leq n$, then the system is in deadlock state. Moreover, if Finish[i] == false, then P_i is deadlocked

Example

- Five processes P_0 through P_4; three resource types; A (7 instances), *B* (2 instances), and *C* (6 instances)
- Snapshot at time T_0:

	Allocation	Request	Available
	A B C	*A B C*	*A B C*
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 0	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

- Sequence <P_0, P_2, P_3, P_1, P_4> will result in Finish[i] = True for all i

Sequence $<P_0, P_2, P_3, P_1, P_4>$

Work = avail = [0 0 0]
Finish[5] = {F, F, F, F, F}

P0
req[0] <= work   Y; [0 0 0] <= [0 0 0]
work = work + alloc[0]   work = [0 0 0] + [0 1 0] = [0 1 0]
Finish[5] = {T, F, F, F, F}

P2
req[2] <= work   Y; [0 0 0] <= [0 1 0]
work = work + alloc[2]   work = [0 1 0] + [3 0 3] = [3 1 3]
Finish[5] = {T, F, T, F, F}

P3
req[3] <= work   Y; [1 0 0] <= [3 1 3]
work = work + alloc[3]   work = [3 1 3] + [2 1 1] = [5 2 4]
Finish[5] = {T, F, T, T, F}

P1
req[1] <= work   Y; [2 0 2] <= [5 2 4]
work = work + alloc[1]   work = [5 2 4] + [2 0 0] = [7 2 4]
Finish[5] = {T, T, T, T, F}

P4
req[4] <= work   Y; [0 0 2] <= [7 2 4]
work = work + alloc[4]   work = [7 2 4] + [0 0 2] = [7 2 6]
Finish[5] = {T, T, T, T, T}

Deadlock Recovery

- Manual intervention
  - Abort all deadlocked processes
- Terminate one or more processes involved in the deadlock
- Preempt resources
  - Selecting a victim process
  - Rollback – return to some safe state, restart process for that state
  - Starvation – same process may always be picked as victim

Completely Fair Scheduler (CFS)

- It is based on Rotating Staircase Deadline Scheduler (RSDL).
- It is default scheduling process since version 2.6.23.
- Elegant handling of I/O and CPU bound process.

- It fairly or equally divides the CPU time among all the processes
- If there are N processes in the ready queue then each process receives (100/N)% of CPU time according to CFS.

Example

Process	Burst Time (ms)
A	10
B	6
C	14
D	6

A = 4
C = 8

A	1	2	3	4	5	6	8	10	
B	1	2	3	4	5	6			
C	1	2	3	4	5	6	8	10	14
D	1	2	3	4	5	6			

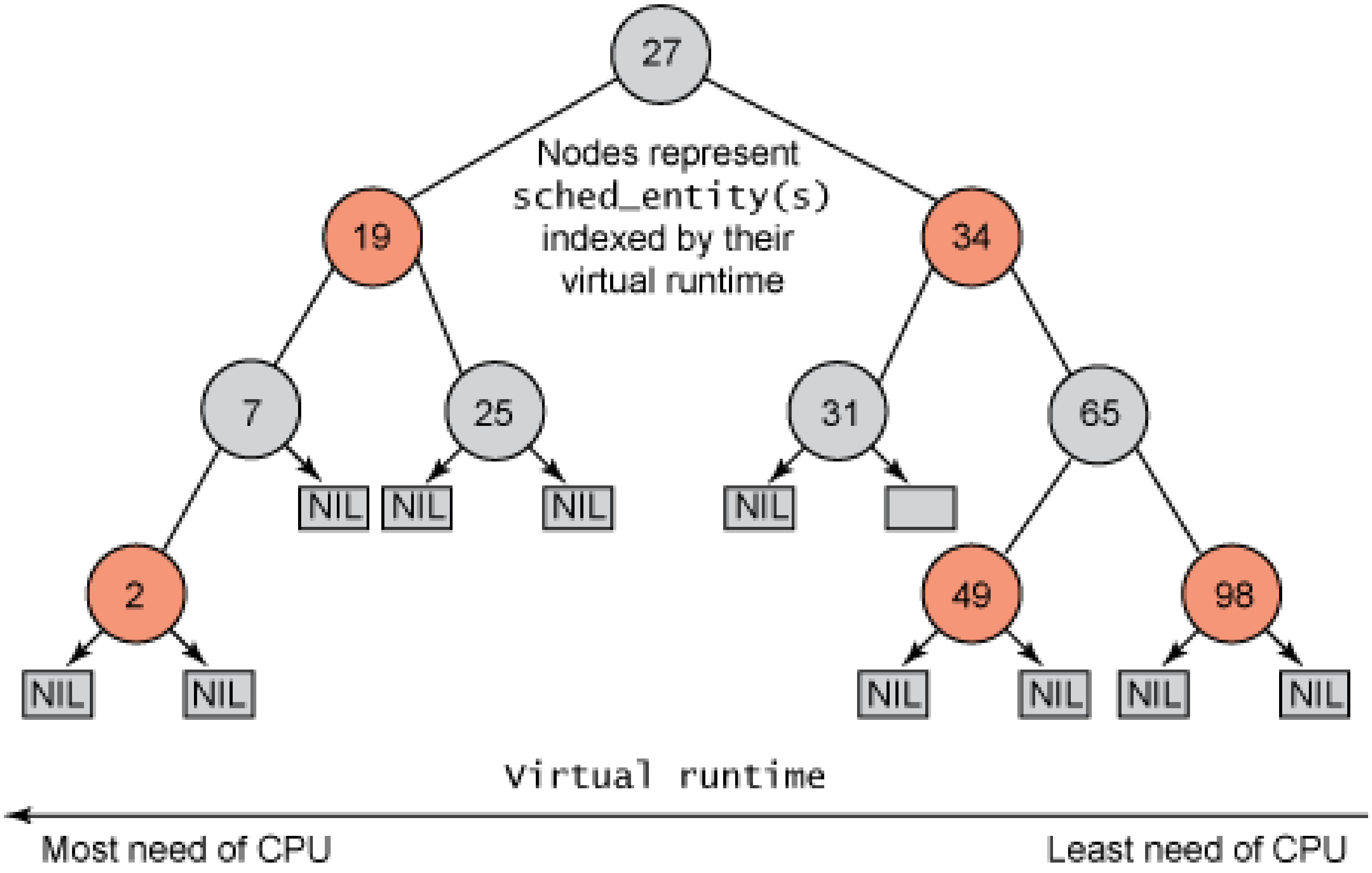
C = 4

- Let time quanta be 4 ms
- Each process gets, 4/4 = 1 ms
- After 6 ms, only two processes left; each process gets 4/2 = 2 ms
- After 10 ms, only process C is left; Process C gets the entire time quanta of 4 ms.

Virtual Runtime

- Each run able process have a virtual time associated with it in Process Control Block (PCB)
- Whenever a context switch happens (or at every scheduling point) then current running process virtual time is increased by virtualruntime_currprocess+=T; where T is time for which it is executed recently.
- Runtime for the process therefore monotonically increases.
- During interrupt, process with the lowest virtual runtime is chosen

implemented using RED-BLACK Trees and not Queues


27
Nodes represent
sched_entity(s)
indexed by their
virtual runtime
19
34
7
25
31
65
NIL
NIL
NIL
NIL
2
49
98
NIL
NIL
NIL
NIL
NIL
NIL
Virtual runtime
Most need of CPU
Least need of CPU

- When the scheduler is invoked to run a new process:
  - The leftmost node of the scheduling tree is chosen and sent for execution
  - If the process simply completes execution, it is removed from the system and scheduling tree
  - If the process reaches its *maximum execution time** or is otherwise stopped (voluntarily or via interrupt) it is reinserted into the scheduling tree based on its new spent *execution time*
  - The new leftmost node will then be selected from the tree, repeating the iteration

** the time the process would have expected to run on an "ideal processor"*

- all the process which are in main memory are inserted into Red Black trees and whenever a new process comes it is inserted into the tree

- During context switch
  - The virtual time for the current process which was executing is updated
  - The new process is decided which has lowest virtual time and it is the left most node of Red Black tree
  - If the current process still has some burst time then it is inserted into the Red Black tree

wait_runtime

- As a process waits for the CPU, the scheduler tracks the amount of time it would have used on the ideal processor
- is used to rank processes for scheduling and to determine the amount of time the process is allowed to execute before being preempted
- The process with the longest wait time is picked by the scheduler and assigned to the CPU
- Process running – wait time decreases

CFS and Priority

- implements priorities by using weighted tasks
  - each task is assigned a weight based on its static priority
- vruntime += t*(weight of a process)
- the task with lower weight (lower-priority) will see time elapse at a faster rate than that of a higher-priority task
- wait_runtime will exhaust more quickly than that of a higher-priority task, so lower-priority tasks will get less CPU time compared to higher-priority tasks

- I/O bound processes (interactive processes) should get higher priority than CPU bound processes (batch processes)
- I/O bound processes -> small CPU bursts -> low vruntime -> appears on the left of RBTree -> higher priority