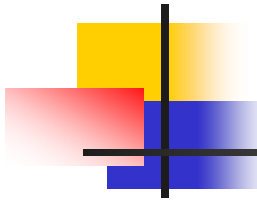
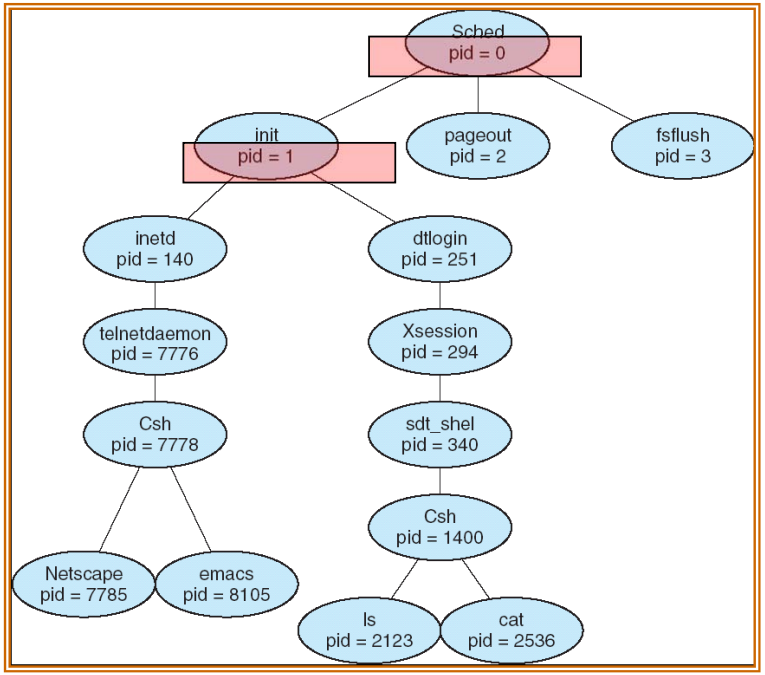


Operations on processes

- Process creation
- Process termination

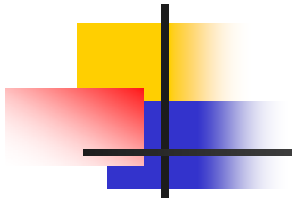
- Parent process create children processes, which, in turn create other processes, forming a tree of processes

PID



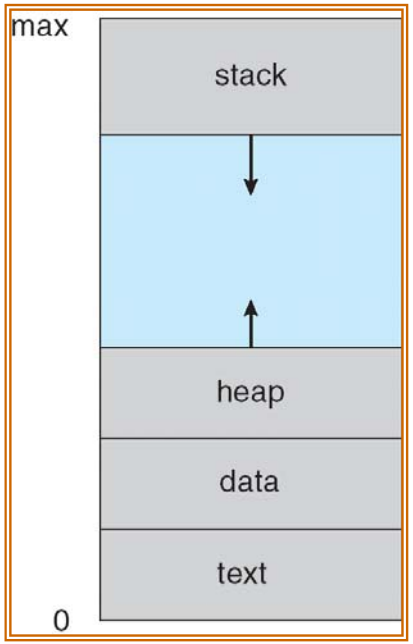
Solaris

- What are the issues?
  - Resource sharing options
    - 1. Parent and child share all resources
    - 2. Child shares subset of parent's resources
    - 3. Parent and child share no resources
  - Execution options
    - 1. Parent and child execute concurrently
    - 2. Parent waits until child terminates
  - Address space
    - 1. Child duplicates parent's address space
    - 2. Child has a program loaded into it



- So, in case of UNIX or Linux
  - fork system call
    - Resource sharing
      - Parent and children share files
      - But CPU time or memory are not shared
    - Execution
      - Parent and children execute concurrently
    - Address space
      - Duplicate but separate address spaces
      - At first, each process has the same user stack, the same local variables, the same heap and the same program
    - Call once, return twice
      - Fork is called by parent, but it returns twice: once to the parent (PID of child) and to the child (0)

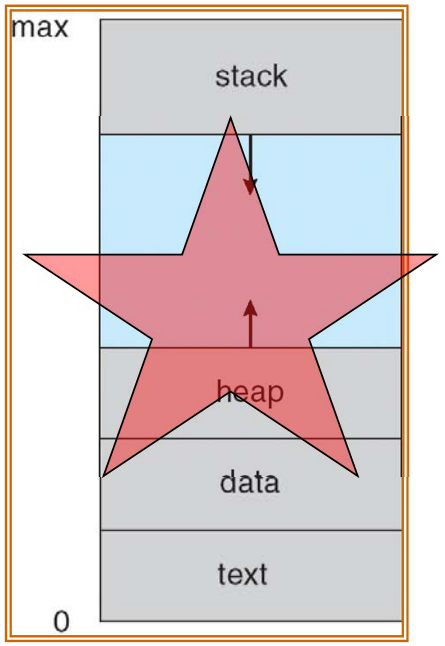
Parent process

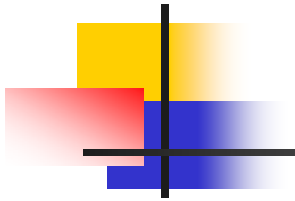


At first, duplicate the address space

But, typically overwrite the new program onto it

Child process





- Every process has the same program !

exec system call is used to load a new program

exec system call loads a program and initializes the program

- Reaping child processes
  - Wait system call

A process waits for its children to terminate

What does "zombie process" mean ?
A terminated process that has not yet been reaped

Call once Return twice

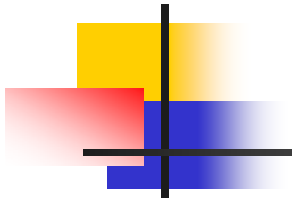
```
int main()
{
    pid_t  pid;
     /* fork another process */
     pid = fork();
     if (pid < 0) { /* error occurred */
              fprintf(stderr, "Fork Failed");
              exit(-1);
     }
     else if (pid == 0) { /* child process */
              execlp("/bin/ls", "ls", NULL);
     }
     else { /* parent process */
              /* parent will wait for the child to complete */
              wait (NULL);
              printf ("Child Complete");
              exit(0);
     }
}
```

```
int main()
{
    pid_t  pid;
        /* fork another process */
        pid = fork();
        if (pid < 0) { /* error occurred */
                fprintf(stderr, "Fork Failed");
                exit(-1);
        }
        else if (pid == 0) { /* child process */
                execlp("/bin/ls", "ls", NULL);
        }
        else { /* parent process */
                /* parent will wait for the child to complete */
                wait (NULL);
                printf ("Child Complete");
                exit(0);
        }
}
```

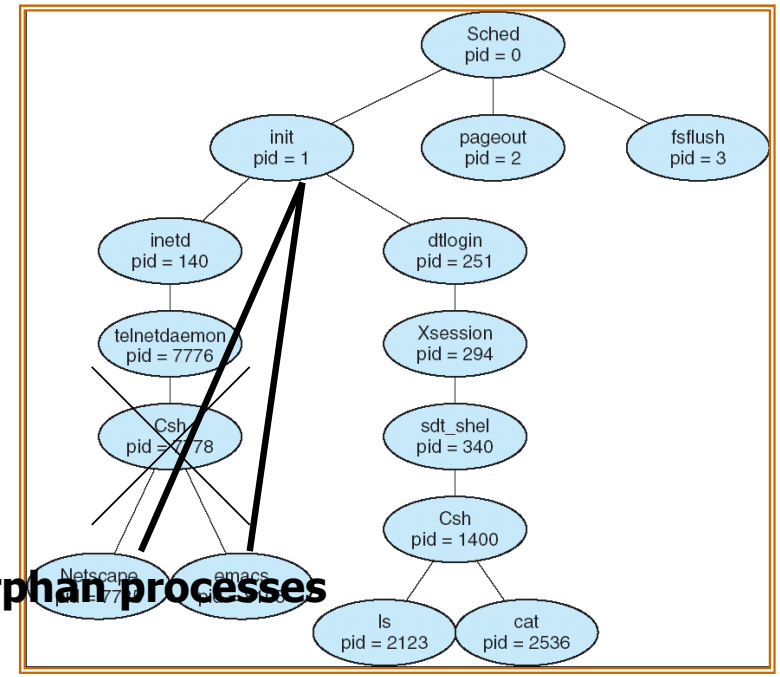
/bin/ls program

parent **child**

PC →



- Process termination
  - Process executes last statement and asks the operating system to delete it (**exit**)
    - Output data from child to parent (via **wait**)
    - Process' resources are deallocated by operating system
  - Parent may terminate execution of children processes (**abort**)
    - Child has exceeded allocated resources
    - Task assigned to child is no longer required
    - If parent is exiting
      - Some operating system do not allow child to continue if its parent terminates
        - All children terminated - *cascading termination*
      - **Orphan process**
        - **What does OS do?**

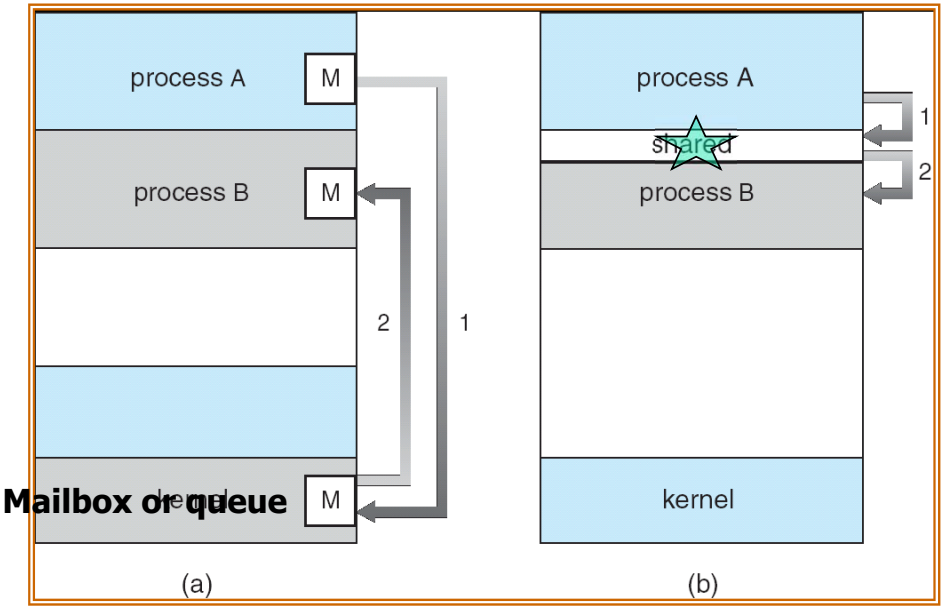


Orphan processes

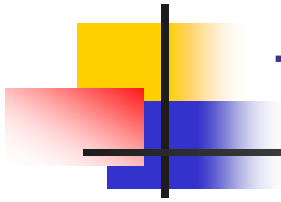
Inter-process communication (IPC)

- **Independent** process cannot affect or be affected by the execution of another process
- **Cooperating** process can affect or be affected by the execution of another process
- Advantages of process cooperation
  - Information sharing
    - e.g. shared file
  - Computation speed-up
    - Parallel processes <if the computer has multiple processing elements>
  - Modularity
  - Convenience
    - e.g. editing printing compiling at once

- IPC
  - 1. Message passing
  - 2. Shared memory

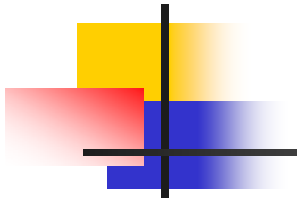


Message passing **Shared memory**



Typical examples of IPC

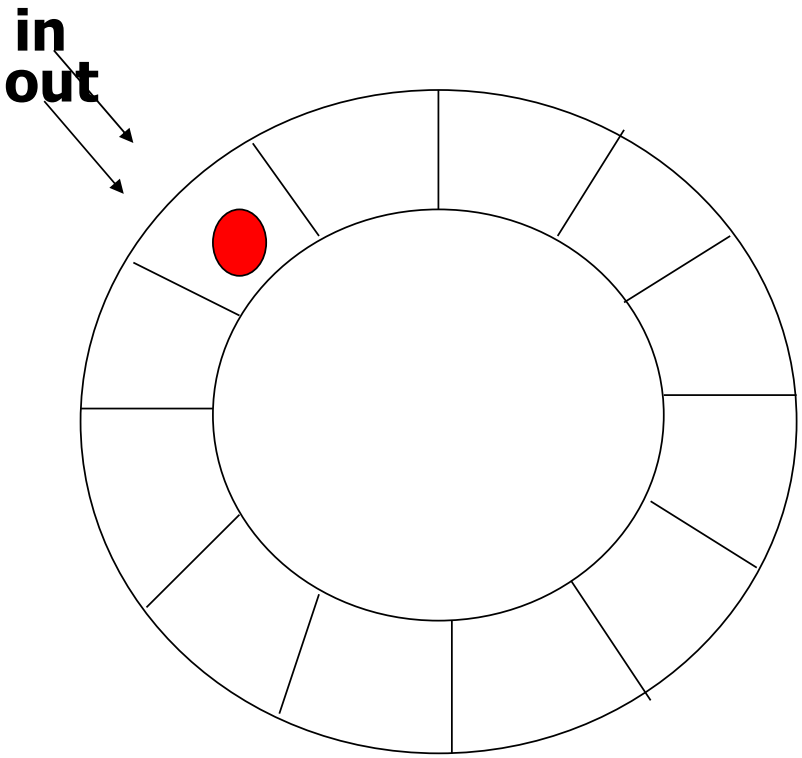
- Producer-Consumer problem
  - Paradigm for cooperating processes, a *producer* process produces information that is consumed by a *consumer* process
    - *unbounded-buffer* places no practical limit on the size of the buffer
    - *bounded-buffer* assumes that there is a fixed buffer size
  - Must be synchronized



- Producer-consumer problem using *a programmed shared-memory*
  - Bounded buffer
    - Shared data

```
#define BUFFER_SIZE 10
typedef struct {
   . . .
} item;

item buffer[BUFFER_SIZE];  // A circular buffer
int in = 0;   // Free position in the buffer
int out = 0;  // The first full position in the buffer
```



Producer

Producing item

Consumer

Waiting
Spin lock

Consuming item

Empty case: in=out

Shared memory, bounded buffer

```
while (true) {
  /* Produce an item */
  while (((in + 1) % BUFFER SIZE)  == out)
    ;   /* do nothing -- no free buffers */
  buffer[in] = item;
  in = (in + 1) % BUFFER SIZE;
}
```

Producer

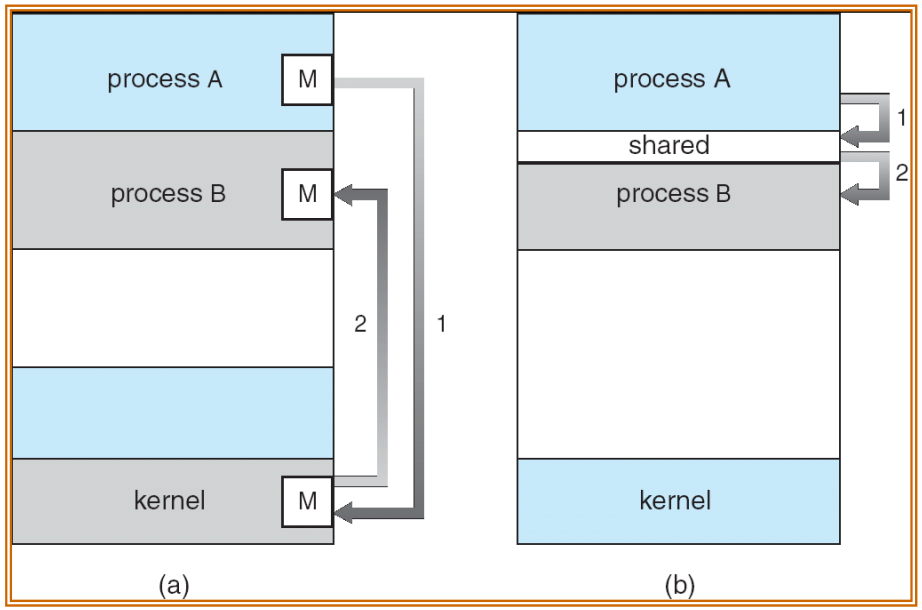
Consumer

```
while (true) {
  while (in == out)
      ; // do nothing -- nothing to consume

  // remove an item from the buffer
  item = buffer[out];
  out = (out + 1) % BUFFER SIZE;
  return item;
}
```

Synchronization !

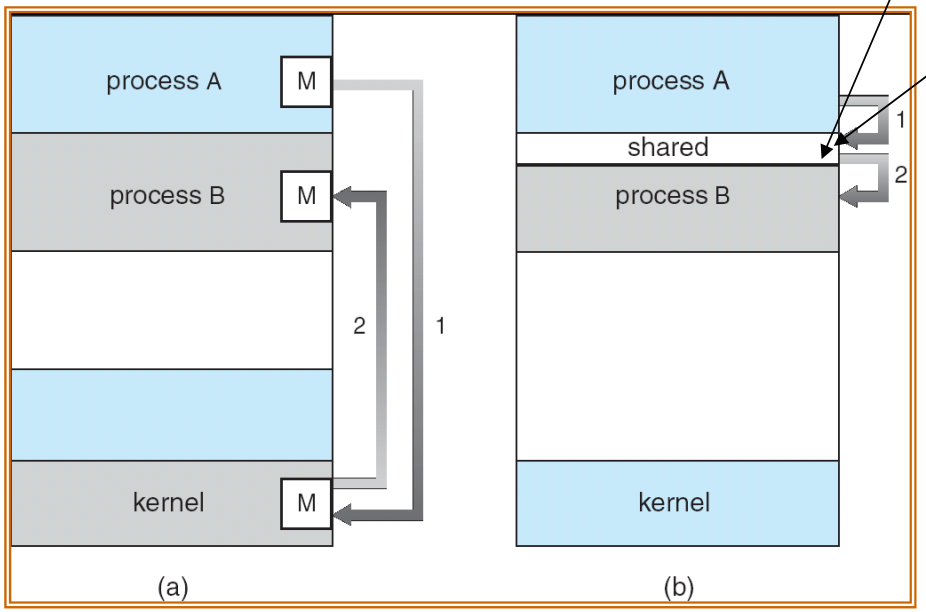
- IPC
  - 1. Message passing
  - 2. Shared memory



Message passing **Shared memory**

- Shared memory

Address space of the process creating the shared memory region



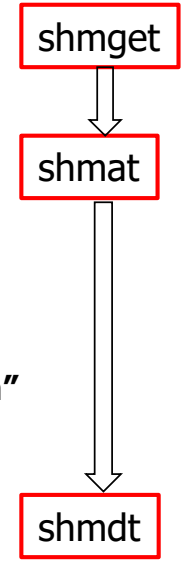
1. Process B creates shared region

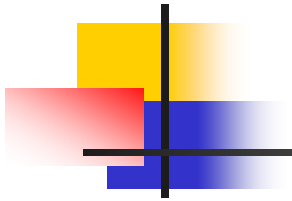
2. Process A attaches the region to its own address space

3. Process B writes "Shared region" in the shared region

4. Process A can read "Shared region" in the shared region

5. Detach a shared region





- Message-passing
  - Processes communicate with each other without resorting to shared variables
  - IPC facility provides two operations:
    - **send**(*message*) – message size fixed or variable
    - **receive**(*message*)
  - If *P* and *Q* wish to communicate, they need to:
    - 1. establish a *communication link* between them
    - 2. exchange messages via send/receive
  - Implementation of communication link
    - physical (e.g., shared memory, hardware bus)
    - logical (e.g., logical properties)

- Establishing links
  - 1. Direct communication
    - Processes must name each other explicitly:
      - **send** (*P*, *message*) – send a message to process P
      - **receive**(*Q*, *message*) – receive a message from process Q

  - 2. Indirect communication
    - Messages are directed and received from mailboxes (also referred to as ports)
      - Each mailbox has a unique id
      - Processes can communicate only if they share a mailbox