Interprocess Communications

IPC allows processes to communicate with one another; it includes messages, shared memory, memory mapping, and semaphores.

Interprocess Communications

IPC allows processes to communicate with one another; it includes messages, shared memory, memory mapping, and semaphores.

Message Queues can be considered to be a resource separate from the programs which utilize them. Processes can write to MQs even if no process is currently reading them (unlike pipes and FIFOs). Messages have “types”, which is an integer. Processes may...

Interprocess Communications

IPC allows processes to communicate with one another; it includes messages, shared memory, memory mapping, and semaphores.

Message Queues can be considered to be a resource separate from the programs which utilize them. Processes can write to MQs even if no process is currently reading them (unlike pipes and FIFOs). Messages have “types”, which is an integer. Processes may...

- retrieve oldest message in the queue (like a FIFO)

Interprocess Communications

IPC allows processes to communicate with one another; it includes messages, shared memory, memory mapping, and semaphores.

Message Queues can be considered to be a resource separate from the programs which utilize them. Processes can write to MQs even if no process is currently reading them (unlike pipes and FIFOs). Messages have “types”, which is an integer. Processes may...

- retrieve oldest message in the queue (like a FIFO)
- retrieve oldest message of a given type

Interprocess Communications

IPC allows processes to communicate with one another; it includes messages, shared memory, memory mapping, and semaphores.

Message Queues can be considered to be a resource separate from the programs which utilize them. Processes can write to MQs even if no process is currently reading them (unlike pipes and FIFOs). Messages have “types”, which is an integer. Processes may...

- retrieve oldest message in the queue (like a FIFO)
- retrieve oldest message of a given type
- retrieve messages by getting the lowest (highest priority) type.

Interprocess Communications

IPC allows processes to communicate with one another; it includes messages, shared memory, memory mapping, and semaphores.

Message Queues can be considered to be a resource separate from the programs which utilize them. Processes can write to MQs even if no process is currently reading them (unlike pipes and FIFOs). Messages have “types”, which is an integer. Processes may...

- retrieve oldest message in the queue (like a FIFO)
- retrieve oldest message of a given type
- retrieve messages by getting the lowest (highest priority) type.

Shared Memory allows processes to share data in a common memory area

Interprocess Communications

IPC allows processes to communicate with one another; it includes messages, shared memory, memory mapping, and semaphores.

Message Queues can be considered to be a resource separate from the programs which utilize them. Processes can write to MQs even if no process is currently reading them (unlike pipes and FIFOs). Messages have “types”, which is an integer. Processes may...

- retrieve oldest message in the queue (like a FIFO)
- retrieve oldest message of a given type
- retrieve messages by getting the lowest (highest priority) type.

Shared Memory allows processes to share data in a common memory area

Memory Mapping is much like shared memory, except the commonality is a file. When the processes are done accessing the memory mapped to a file, the file still exists and may subsequently be re-used.

Interprocess Communications

IPC allows processes to communicate with one another; it includes messages, shared memory, memory mapping, and semaphores.

Message Queues can be considered to be a resource separate from the programs which utilize them. Processes can write to MQs even if no process is currently reading them (unlike pipes and FIFOs). Messages have “types”, which is an integer. Processes may...

- retrieve oldest message in the queue (like a FIFO)
- retrieve oldest message of a given type
- retrieve messages by getting the lowest (highest priority) type.

Shared Memory allows processes to share data in a common memory area

Memory Mapping is much like shared memory, except the commonality is a file. When the processes are done accessing the memory mapped to a file, the file still exists and may subsequently be re-used.

Semaphores allow processes to coordinate access to resources, especially including shared memory and memory mapped to a file.

Generic IPC Functions

- IPC resources use an “IPC descriptor”, akin to a file descriptor

Generic IPC Functions

- IPC resources use an “IPC descriptor”, akin to a file descriptor
- In IPC, these descriptors are called **keys**; but they’re mostly up to you, the programmer, to choose.

Generic IPC Functions

- IPC resources use an “IPC descriptor”, akin to a file descriptor
- In IPC, these descriptors are called **keys**; but they’re mostly up to you, the programmer, to choose.
- The program name/file itself may be used to generate a key, along with a “project identifier”

Generic IPC Functions

- IPC resources use an “IPC descriptor”, akin to a file descriptor
- In IPC, these descriptors are called **keys**; but they’re mostly up to you, the programmer, to choose.
- The program name/file itself may be used to generate a key, along with a “project identifier”

```
#include <sys/types.h>
#include <sys/ipc.h>
key_t ftok(const char *pathname, int proj_id);
```

- ftok() generates a key which *identifies* an IPC.

Generic IPC Functions

- IPC resources use an “IPC descriptor”, akin to a file descriptor
- In IPC, these descriptors are called **keys**; but they’re mostly up to you, the programmer, to choose.
- The program name/file itself may be used to generate a key, along with a “project identifier”

```
#include <sys/types.h>
#include <sys/ipc.h>
key_t ftok(const char *pathname, int proj_id);
```

- ftok() generates a key which *identifies* an IPC.
- The ftok() function generates a System V type IPC key, which may be used with msgget(), semget(), or shmget().

Generic IPC Functions

- IPC resources use an “IPC descriptor”, akin to a file descriptor
- In IPC, these descriptors are called **keys**; but they’re mostly up to you, the programmer, to choose.
- The program name/file itself may be used to generate a key, along with a “project identifier”

```
#include <sys/types.h>
#include <sys/ipc.h>
key_t ftok(const char *pathname, int proj_id);
```

- ftok() generates a key which *identifies* an IPC.
- The ftok() function generates a System V type IPC key, which may be used with msgget(), semget(), or shmget().
- Multiple processes may use the same filename to generate a key via ftok(), with up to 256 “projects” via proj_id. (often argv[0])

Generic IPC Flags

The various system calls that set up an IPC identifier (...*get()*) from a key take some common flags:

IPC_PRIVATE unique IPC channel (good for forks)

1. Create a new IPC entity and return an identifier for it
   (used by processes possessing a common ancestor; ie. related via a fork() call)

Generic IPC Flags

The various system calls that set up an IPC identifier (...*get()*) from a key take some common flags:

IPC_PRIVATE unique IPC channel (good for forks)

1. Create a new IPC entity and return an identifier for it
   (used by processes possessing a common ancestor; ie. related via a fork() call)
2. Child can pass the identifier along to a subsequent program via an argument in an exec family call

IPC_CREAT

Generic IPC Flags

The various system calls that set up an IPC identifier (...*get()*) from a key take some common flags:

IPC_PRIVATE unique IPC channel (good for forks)

1. Create a new IPC entity and return an identifier for it
   (used by processes possessing a common ancestor; ie. related via a fork() call)
2. Child can pass the identifier along to a subsequent program via an argument in an exec family call

IPC_CREAT

1. if the IPC entity does not exist, create a new one and return an identifier for it

Generic IPC Flags

The various system calls that set up an IPC identifier (...*get()*) from a key take some common flags:

IPC_PRIVATE unique IPC channel (good for forks)

1. Create a new IPC entity and return an identifier for it
   (used by processes possessing a common ancestor; ie. related via a fork() call)
2. Child can pass the identifier along to a subsequent program via an argument in an exec family call

IPC_CREAT

1. if the IPC entity does not exist, create a new one and return an identifier for it
2. if the IPC entity exists already, return an identifier for it

IPC_CREAT|IPC_EXCL

Generic IPC Flags

The various system calls that set up an IPC identifier (...*get()*) from a key take some common flags:

IPC_PRIVATE unique IPC channel (good for forks)

1. Create a new IPC entity and return an identifier for it
   (used by processes possessing a common ancestor; ie. related via a fork() call)
2. Child can pass the identifier along to a subsequent program via an argument in an exec family call

IPC_CREAT

1. if the IPC entity does not exist, create a new one and return an identifier for it
2. if the IPC entity exists already, return an identifier for it

IPC_CREAT|IPC_EXCL

1. if the identifier does not exist, create a new one

Generic IPC Flags

The various system calls that set up an IPC identifier (...*get()*) from a key take some common flags:

IPC_PRIVATE unique IPC channel (good for forks)

1. Create a new IPC entity and return an identifier for it
   (used by processes possessing a common ancestor; ie. related via a fork() call)
2. Child can pass the identifier along to a subsequent program via an argument in an exec family call

IPC_CREAT

1. if the IPC entity does not exist, create a new one and return an identifier for it
2. if the IPC entity exists already, return an identifier for it

IPC_CREAT|IPC_EXCL

1. if the identifier does not exist, create a new one
2. if the identifier exists already, return error (-1/EACCESS)

   (note: does not exclude access to other processes)

IPC Permissions

Each IPC has a full set of permissions available:

```
struct ipc_perm {
  key_t  key;          the associated IPC key
  ushort uid;          owner's user id
  ushort gid;          owner's group id
  ushort cuid;         creator's user id
  ushort cgid;         creator's group id
  ushort mode;         access control
  ushort seq;          slot usage sequence number
}
```

- The mode is an octal access mode, akin to that used for open().

IPC Permissions

Each IPC has a full set of permissions available:

```
struct ipc_perm {
  key_t  key;          the associated IPC key
  ushort uid;          owner's user id
  ushort gid;          owner's group id
  ushort cuid;         creator's user id
  ushort cgid;         creator's group id
  ushort mode;         access control
  ushort seq;          slot usage sequence number
}
```

- The mode is an octal access mode, akin to that used for open().
- Each time an IPC object is closed by the system, the seq value gets incremented by the maximum qty of IPC objects that may reside in the system.

IPC Permissions

Each IPC has a full set of permissions available:

```
struct ipc_perm {
  key_t  key;          the associated IPC key
  ushort uid;          owner's user id
  ushort gid;          owner's group id
  ushort cuid;         creator's user id
  ushort cgid;         creator's group id
  ushort mode;         access control
  ushort seq;          slot usage sequence number
}
```

- The mode is an octal access mode, akin to that used for open().
- Each time an IPC object is closed by the system, the seq value gets incremented by the maximum qty of IPC objects that may reside in the system.
- You won't need to be concerned about seq!

Message Queues: msgget()

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgget(key_t key, int msgflg);
```

- A *message queue* is a linked list of messages stored by the kernel (like a FIFO)

Message Queues: msgget()

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgget(key_t key, int msgflg);
```

- A *message queue* is a linked list of messages stored by the kernel (like a FIFO)
- Messages may have varying lengths

Message Queues: msgget()

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgget(key_t key, int msgflg);
```

- A *message queue* is a linked list of messages stored by the kernel (like a FIFO)
- Messages may have varying lengths
- MQs provide communications between processes on a single machine

Message Queues: msgget()

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgget(key_t key, int msgflg);
```

- A *message queue* is a linked list of messages stored by the kernel (like a FIFO)
- Messages may have varying lengths
- MQs provide communications between processes on a single machine
- The msgget() function *gets* a new MQ or provides access to an existing MQ

Message Queues: msgget()

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgget(key_t key, int msgflg);
```

- A *message queue* is a linked list of messages stored by the kernel (like a FIFO)
- Messages may have varying lengths
- MQs provide communications between processes on a single machine
- The msgget() function *gets* a new MQ or provides access to an existing MQ
- New messages are prepended to the MQ by msgsnd()

Message Queues: msgget()

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgget(key_t key, int msgflg);
```

- A *message queue* is a linked list of messages stored by the kernel (like a FIFO)
- Messages may have varying lengths
- MQs provide communications between processes on a single machine
- The msgget() function *gets* a new MQ or provides access to an existing MQ
- New messages are prepended to the MQ by msgsnd()
- Oldest messages are taken off the MQ by msgrcv()

Message Queues: msgget()

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgget(key_t key, int msgflg);
```

- A *message queue* is a linked list of messages stored by the kernel (like a FIFO)
- Messages may have varying lengths
- MQs provide communications between processes on a single machine
- The msgget() function *gets* a new MQ or provides access to an existing MQ
- New messages are prepended to the MQ by msgsnd()
- Oldest messages are taken off the MQ by msgrcv()
- flag: this is where you choose IPC_PRIVATE, IPC_CREATE, or IPC_CREAT|IPC_EXCL.

Message Queues: msgget()

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgget(key_t key, int msgflg);
```

- A *message queue* is a linked list of messages stored by the kernel (like a FIFO)
- Messages may have varying lengths
- MQs provide communications between processes on a single machine
- The msgget() function *gets* a new MQ or provides access to an existing MQ
- New messages are prepended to the MQ by msgsnd()
- Oldest messages are taken off the MQ by msgrcv()
- flag: this is where you choose IPC_PRIVATE, IPC_CREATE, or IPC_CREAT|IPC_EXCL.
- Success: returns a MQ identifier. Failure: returns -1.

Message Queues: msgget()

- To create a new message queue:

  int msgqid= msgget(-key-,IPC_CREAT|IPC_EXCL)

Message Queues: msgget()

- To create a new message queue:

  int msgqid= msgget(-key-,IPC_CREAT|IPC_EXCL)

- To access an already existing message queue:

  int msgqid= msgget(-key-,IPC_CREAT)

Message Queues: msgget()

- To create a new message queue:

  int msgqid= msgget(-key-,IPC_CREAT|IPC_EXCL)

- To access an already existing message queue:

  int msgqid= msgget(-key-,IPC_CREAT)

- If the resulting msgqid is -1, an error has occurred; see errno

Message Queues: msgsnd()

int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg);

- Messages to be sent must begin with four bytes; the “message type”. Example:

```
struct MsgBuf_str {
        long msgtyp;            // message type
        char msgtxt[MSGMAX];    // your message data, need not be a string
        }
```

Message Queues: msgsnd()

int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg);

- Messages to be sent must begin with four bytes; the “message type”. Example:

```
struct MsgBuf_str {
        long msgtyp;            // message type
        char msgtxt[MSGMAX];    // your message data, need not be a string
        }
```

- This call appends a message to the MQ specified by msqid

Message Queues: msgsnd()

int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg);

- Messages to be sent must begin with four bytes; the “message type”. Example:

```
struct MsgBuf_str {
        long msgtyp;            // message type
        char msgtxt[MSGMAX];    // your message data, need not be a string
        }
```

- This call appends a message to the MQ specified by msqid
- msgsz length of message

Message Queues: msgsnd()

int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg);

- Messages to be sent must begin with four bytes; the “message type”. Example:

```
struct MsgBuf_str {
        long msgtyp;            // message type
        char msgtxt[MSGMAX];    // your message data, need not be a string
        }
```

- This call appends a message to the MQ specified by msqid
- msgsz length of message
- msgflg=0: process may block

  msgflg=IPC_NOWAIT: msgsnd will fail (-1/EACCESS) if it otherwise would have blocked

Message Queues: msgsnd()

int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg);

- Messages to be sent must begin with four bytes; the “message type”. Example:

```
struct MsgBuf_str {
        long msgtyp;            // message type
        char msgtxt[MSGMAX];    // your message data, need not be a string
        }
```

- This call appends a message to the MQ specified by msqid
- msgsz length of message
- msgflg=0: process may block

  msgflg=IPC_NOWAIT: msgsnd will fail (-1/EACCESS) if it otherwise would have blocked
- Messages of zero length (ie. they have no msgtxt) are permitted

Message Queues: msgsnd()

int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg);

- Messages to be sent must begin with four bytes; the “message type”. Example:

```
struct MsgBuf_str {
        long msgtyp;            // message type
        char msgtxt[MSGMAX];    // your message data, need not be a string
        }
```

- This call appends a message to the MQ specified by msqid
- msgsz length of message
- msgflg=0: process may block

  msgflg=IPC_NOWAIT: msgsnd will fail (-1/EACCESS) if it otherwise would have blocked
- Messages of zero length (ie. they have no msgtxt) are permitted
- Example:

```
struct MsgBuf_str msgbuf;
bytes_sent= msgsnd(msqid,&msgbuf,sizeof(struct MsgBuf_str),0);
```

(msgtyp discussed on next slide)

Message Queues: msgrcv()

ssize_t msgrcv(int msqid, void *msgp, size_t msgsz, long msgtyp,int msgflg);

- The first four bytes of a message specify its *type* *(msgtyp)*

Message Queues: msgrcv()

ssize_t msgrcv(int msqid, void *msgp, size_t msgsz, long msgtyp,int msgflg);

- The first four bytes of a message specify its *type* *(msgtyp)*
- This call removes a message from the MQ specified by msqid

Message Queues: msgrcv()

ssize_t msgrcv(int msqid, void *msgp, size_t msgsz, long msgtyp,int msgflg);

- The first four bytes of a message specify its *type* *(msgtyp)*
- This call removes a message from the MQ specified by msqid
- One may use the message type to specify where it came from, what the sender wants done with it, for message typing, or for priority.

Message Queues: msgrcv()

ssize_t msgrcv(int msqid, void *msgp, size_t msgsz, long msgtyp,int msgflg);

- The first four bytes of a message specify its *type* *(msgtyp)*
- This call removes a message from the MQ specified by msqid
- One may use the message type to specify where it came from, what the sender wants done with it, for message typing, or for priority.

  msgtyp $= 0$: oldest message on MQ received

Message Queues: msgrcv()

ssize_t msgrcv(int msqid, void *msgp, size_t msgsz, long msgtyp,int msgflg);

- The first four bytes of a message specify its *type* *(msgtyp)*
- This call removes a message from the MQ specified by msqid
- One may use the message type to specify where it came from, what the sender wants done with it, for message typing, or for priority.

msgtyp $= 0$: oldest message on MQ received

> 0 : oldest message with type of msgtyp received

Message Queues: msgrcv()

ssize_t msgrcv(int msqid, void *msgp, size_t msgsz, long msgtyp,int msgflg);

- The first four bytes of a message specify its *type* *(msgtyp)*
- This call removes a message from the MQ specified by msqid
- One may use the message type to specify where it came from, what the sender wants done with it, for message typing, or for priority.

msgtyp $= 0$: oldest message on MQ received

> 0 : oldest message with type of msgtyp received

< 0 : oldest messsage with $\min(type) \leq |msgtyp|$ received

: *(supports priority queueing)*

Message Queues: msgrcv()

ssize_t msgrcv(int msqid, void *msgp, size_t msgsz, long msgtyp,int msgflg);

- The first four bytes of a message specify its *type* *(msgtyp)*
- This call removes a message from the MQ specified by msqid
- One may use the message type to specify where it came from, what the sender wants done with it, for message typing, or for priority.

  msgtyp $= 0$: oldest message on MQ received

  > 0 : oldest message with type of msgtyp received

  < 0 : oldest messsage with $\min(type) \leq |msgtyp|$ received

  : *(supports priority queueing)*

- msgflg is as for msgsnd()

Message Queues: msgrcv()

ssize_t msgrcv(int msqid, void *msgp, size_t msgsz, long msgtyp,int msgflg);

- The first four bytes of a message specify its *type* *(msgtyp)*
- This call removes a message from the MQ specified by msqid
- One may use the message type to specify where it came from, what the sender wants done with it, for message typing, or for priority.

  msgtyp $= 0$: oldest message on MQ received

  > 0 : oldest message with type of msgtyp received

  < 0 : oldest messsage with $\min(type) \leq |msgtyp|$ received

  : *(supports priority queueing)*

- msgflg is as for msgsnd()
- The msgsnd() and msgrcv() functions use a single MQ with positive types acting like multiple queues (multiplexing).

Message Queues: msgctl()

int msgctl(int msqid, int cmd, struct msqid_ds *mbufp);

- msqid specifies which MQ will be “controlled”

Message Queues: msgctl()

int msgctl(int msqid, int cmd, struct msqid_ds *mbufp);

- msqid specifies which MQ will be “controlled”
- Performs the cmd as selected from:

Message Queues: msgctl()

int msgctl(int msqid, int cmd, struct msqid_ds *mbufp);

- msqid specifies which MQ will be “controlled”
- Performs the cmd as selected from:

  IPC_RMID (mbufp=NULL) if calling process was creator of MQ, or has been granted privileges: destroys the MQ

Message Queues: msgctl()

int msgctl(int msqid, int cmd, struct msqid_ds *mbufp);

- msqid specifies which MQ will be “controlled”
- Performs the cmd as selected from:

IPC_RMID	(mbufp=NULL) if calling process was creator of MQ, or has been granted privileges: destroys the MQ
IPC_SET	creator/grantee of privileges/superuser may grant privileges to another process using mbuf.uid, mbuf.gid, and mbuf.mode permissions.

Message Queues: msgctl()

int msgctl(int msqid, int cmd, struct msqid_ds *mbufp);

- msqid specifies which MQ will be “controlled”
- Performs the cmd as selected from:

IPC_RMID	(mbufp=NULL) if calling process was creator of MQ, or has been granted privileges: destroys the MQ
IPC_SET	creator/grantee of privileges/superuser may grant privileges to another process using mbuf.uid, mbuf.gid, and mbuf.mode permissions.
IPC_STAT	copy MQ’s current msqid_ds into *mbufp

Message Queues: struct msqid_ds

```
struct msqid_ds {
        struct ipc_perm    msg_perm;     // Ownership and permissions
        time_t             msg_stime;    // Time of last msgsnd
        time_t             msg_rtime;    // Time of last msgrcv
        time_t             msg_ctime;    // Time of last change
        unsigned long      __msg_cbytes; // Current qty of bytes in queue (non-standard)
        msgqnum_t          msg_qnum;     // Current qty of messages in queue
        msglen_t           msg_qbytes;   // Maximum qty of bytes allowed in queue
        pid_t              msg_lspid;    // PID of last msgsnd
        pid_t              msg_lrpid;    // PID of last msgrcv
        };
```

- This is what you get with msgctl() + IPC_STAT

Message Queues: struct msqid_ds

```
struct msqid_ds {
        struct ipc_perm    msg_perm;     // Ownership and permissions
        time_t             msg_stime;    // Time of last msgsnd
        time_t             msg_rtime;    // Time of last msgrcv
        time_t             msg_ctime;    // Time of last change
        unsigned long      __msg_cbytes; // Current qty of bytes in queue (non-standard)
        msgqnum_t          msg_qnum;     // Current qty of messages in queue
        msglen_t           msg_qbytes;   // Maximum qty of bytes allowed in queue
        pid_t              msg_lspid;    // PID of last msgsnd
        pid_t              msg_lrpid;    // PID of last msgrcv
        };
```

- This is what you get with msgctl() + IPC_STAT
- Note: there may be gaps between entries to make MQs the same length on different architectures.

Semaphores

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
```

- Semaphores allow processes to control access to resources. Typical usage:

Semaphores

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
```

- Semaphores allow processes to control access to resources. Typical usage:
  - test if a semaphore is set

Semaphores

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
```

- Semaphores allow processes to control access to resources. Typical usage:
  - test if a semaphore is set
  - if not set, set the semaphore (ie. obtain access to resource)

Semaphores

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
```

- Semaphores allow processes to control access to resources. Typical usage:
  - test if a semaphore is set
  - if not set, set the semaphore (ie. obtain access to resource)
  - act upon the resource

Semaphores

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
```

- Semaphores allow processes to control access to resources. Typical usage:
  - test if a semaphore is set
  - if not set, set the semaphore (ie. obtain access to resource)
  - act upon the resource
  - release the semaphore (ie. unset it)

Semaphores

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
```

- Semaphores allow processes to control access to resources. Typical usage:
  - test if a semaphore is set
  - if not set, set the semaphore (ie. obtain access to resource)
  - act upon the resource
  - release the semaphore (ie. unset it)
- Use these to coordinate/synchronize multiple processes (or threads)

Semaphores

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
```

- Semaphores allow processes to control access to resources. Typical usage:
  - test if a semaphore is set
  - if not set, set the semaphore (ie. obtain access to resource)
  - act upon the resource
  - release the semaphore (ie. unset it)
- Use these to coordinate/synchronize multiple processes (or threads)
- Common resources include IPC such as message queues, shared memory, and memory mapped files, as well as hardware (sound channels, printer port, serial ports, etc).

Semaphores

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
```

- Semaphores allow processes to control access to resources. Typical usage:
  - test if a semaphore is set
  - if not set, set the semaphore (ie. obtain access to resource)
  - act upon the resource
  - release the semaphore (ie. unset it)
- Use these to coordinate/synchronize multiple processes (or threads)
- Common resources include IPC such as message queues, shared memory, and memory mapped files, as well as hardware (sound channels, printer port, serial ports, etc).
- Semaphores are **atomic**: they perform a *test and set* pair of operations that cannot be interrupted by the kernel

Semaphores

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
```

- Semaphores allow processes to control access to resources. Typical usage:
  - test if a semaphore is set
  - if not set, set the semaphore (ie. obtain access to resource)
  - act upon the resource
  - release the semaphore (ie. unset it)
- Use these to coordinate/synchronize multiple processes (or threads)
- Common resources include IPC such as message queues, shared memory, and memory mapped files, as well as hardware (sound channels, printer port, serial ports, etc).
- Semaphores are **atomic**: they perform a *test and set* pair of operations that cannot be interrupted by the kernel
- System-V style semaphores are obtained as a *set* of semaphores, rather than one at a time

Semaphores

- Basic semaphore usage acts akin to:

```
if(block until semaphore is “not set”) {
        set semaphore
        }
```

Semaphores

- Basic semaphore usage acts akin to:

```
if(block until semaphore is “not set”) {
        set semaphore
        }
```

- Except that *no interrupts are permitted between the test (if...) and the setting of the semaphore*. This includes process switching interrupts.

Semaphores

- Basic semaphore usage acts akin to:

```
if(block until semaphore is “not set”) {
        set semaphore
        }
```

- Except that *no interrupts are permitted between the test (if...) and the setting of the semaphore*. This includes process switching interrupts.
- This non-interruptability is what is meant by an *atomic* action

Semaphores: semget()

int semget(key_t key, int nsems, int semflg);

- semget() returns a set of semaphores with an identifier *(aka semid)* to be associated with the IPC key

Semaphores: semget()

int semget(key_t key, int nsems, int semflg);

- semget() returns a set of semaphores with an identifier *(aka semid)* to be associated with the IPC key
- key is the IPC key

Semaphores: semget()

int semget(key_t key, int nsems, int semflg);

- semget() returns a set of semaphores with an identifier *(aka semid)* to be associated with the IPC key
- key is the IPC key
- nsems is the number (quantity) of semaphores requested in the set

Semaphores: semget()

int semget(key_t key, int nsems, int semflg);

- semget() returns a set of semaphores with an identifier *(aka semid)* to be associated with the IPC key
- key is the IPC key
- nsems is the number (quantity) of semaphores requested in the set
- Use semflg=IPC_CREAT|IPC_EXCL to *create* a new semaphore set

  The least significant 9 bits of the semflg defines permissions (user, group, other)

Semaphores: semget()

int semget(key_t key, int nsems, int semflg);

- semget() returns a set of semaphores with an identifier *(aka semid)* to be associated with the IPC key
- key is the IPC key
- nsems is the number (quantity) of semaphores requested in the set
- Use semflg=IPC_CREAT|IPC_EXCL to *create* a new semaphore set
  The least significant 9 bits of the semflg defines permissions (user, group, other)
- Use semflg=IPC_CREAT to *access* (share) an existing semaphore set

Semaphores: semget()

int semget(key_t key, int nsems, int semflg);

- semget() returns a set of semaphores with an identifier *(aka semid)* to be associated with the IPC key
- key is the IPC key
- nsems is the number (quantity) of semaphores requested in the set
- Use semflg=IPC_CREAT|IPC_EXCL to *create* a new semaphore set
  The least significant 9 bits of the semflg defines permissions (user, group, other)
- Use semflg=IPC_CREAT to *access* (share) an existing semaphore set
- Values of a newly created set of semaphores are indeterminate.

Semaphores: semget()

int semget(key_t key, int nsems, int semflg);

- semget() returns a set of semaphores with an identifier *(aka semid)* to be associated with the IPC key
- key is the IPC key
- nsems is the number (quantity) of semaphores requested in the set
- Use semflg=IPC_CREAT|IPC_EXCL to *create* a new semaphore set
  The least significant 9 bits of the semflg defines permissions (user, group, other)
- Use semflg=IPC_CREAT to *access* (share) an existing semaphore set
- Values of a newly created set of semaphores are indeterminate.
- SEMMNI limits the total number of semaphore sets
  (on my home Linux system, SEMMNI=128)

Semaphores: semget()

int semget(key_t key, int nsems, int semflg);

- semget() returns a set of semaphores with an identifier *(aka semid)* to be associated with the IPC key
- key is the IPC key
- nsems is the number (quantity) of semaphores requested in the set
- Use semflg=IPC_CREAT|IPC_EXCL to *create* a new semaphore set
  The least significant 9 bits of the semflg defines permissions (user, group, other)
- Use semflg=IPC_CREAT to *access* (share) an existing semaphore set
- Values of a newly created set of semaphores are indeterminate.
- SEMMNI limits the total number of semaphore sets
  (on my home Linux system, SEMMNI=128)
- SEMMSL is the maximum number of semaphores per semid
  (on my home Linux system, SEMMSL=250)

Semaphores: semctl()

int semctl(int semid, int semnum, int cmd, ...);

- semctl(), depending on the cmd, *atomically* operates on all the semaphores in the set or on the $semnum^{th}$ semaphore.

Semaphores: semctl()

int semctl(int semid, int semnum, int cmd, ...);

- semctl(), depending on the cmd, *atomically* operates on all the semaphores in the set or on the $semnum^{th}$ semaphore.
- Semaphores are numbered starting at 0

Semaphores: semctl()

int semctl(int semid, int semnum, int cmd, ...);

- semctl(), depending on the cmd, *atomically* operates on all the semaphores in the set or on the $semnum^{th}$ semaphore.
- Semaphores are numbered starting at 0
- This function has three or four arguments (depending on cmd)

Semaphores: semctl()

int semctl(int semid, int semnum, int cmd, ...);

- semctl(), depending on the cmd, *atomically* operates on all the semaphores in the set or on the $semnum^{th}$ semaphore.
- Semaphores are numbered starting at 0
- This function has three or four arguments (depending on cmd)
- The fourth argument, when used, is of the type union semun:

```
union semun {
        unsigned short  *array;  //Array for GETALL, SETALL
        int              val;    //Value for SETVAL
        struct semid_ds *buf;    //Buffer for IPC_STAT, IPC_SET
        unsigned short  *array;  //Array for GETALL, SETALL
        struct seminfo  *__buf;  //Buffer for IPC_INFO
        }
```

Semaphores: semctl(), con't.

cmd	*returns*
GETVAL	value of semid:semnum semaphore
SETVAL	sets semid:semnum semaphore to val
GETALL	places semvals into array
SETALL	sets semaphores using array
GETNCNT	returns value of semncnt
GETZCNT	returns value of semzcnt
IPC_RMID	remove (delete) semaphore set

(semzcnt is the qty of processes that are currently blocked, waiting for the semnum semaphore to become zero)

Semaphores: semop()

int semop(int semid, struct sembuf *sops, unsigned nsops);

- semop() *atomically* performs an array of semaphore operations

  (ie. not-interruptible, as if it were one instruction)

Semaphores: semop()

int semop(int semid, struct sembuf *sops, unsigned nsops);

- semop() *atomically* performs an array of semaphore operations

  (ie. not-interruptible, as if it were one instruction)

- Each semaphore has the following values associated with it:

Semaphores: semop()

int semop(int semid, struct sembuf *sops, unsigned nsops);

- semop() *atomically* performs an array of semaphore operations

  (ie. not-interruptible, as if it were one instruction)

- Each semaphore has the following values associated with it:

```
unsigned short  semval;   // semaphore's value
```

Semaphores: semop()

int semop(int semid, struct sembuf *sops, unsigned nsops);

- semop() *atomically* performs an array of semaphore operations

  (ie. not-interruptible, as if it were one instruction)

- Each semaphore has the following values associated with it:

```
unsigned short   semval;    // semaphore's value
unsigned short   semzcnt;   // qty processes waiting for this semaphore to go to zero
```

Semaphores: semop()

int semop(int semid, struct sembuf *sops, unsigned nsops);

- semop() *atomically* performs an array of semaphore operations

  (ie. not-interruptible, as if it were one instruction)

- Each semaphore has the following values associated with it:

```
unsigned short   semval;    // semaphore's value
unsigned short   semzcnt;   // qty processes waiting for this semaphore to go to zero
unsigned short   semncnt;   // qty processes waiting for this semaphore to increase value
```

Semaphores: semop()

int semop(int semid, struct sembuf *sops, unsigned nsops);

- semop() *atomically* performs an array of semaphore operations

  (ie. not-interruptible, as if it were one instruction)

- Each semaphore has the following values associated with it:

```
unsigned short   semval;    // semaphore's value
unsigned short   semzcnt;   // qty processes waiting for this semaphore to go to zero
unsigned short   semncnt;   // qty processes waiting for this semaphore to increase value
pid_t            sempid;    // process id that did the last operation
```

Semaphores: semop()

int semop(int semid, struct sembuf *sops, unsigned nsops);

- semop() *atomically* performs an array of semaphore operations

  (ie. not-interruptible, as if it were one instruction)

- Each semaphore has the following values associated with it:

```
unsigned short   semval;    // semaphore's value
unsigned short   semzcnt;   // qty processes waiting for this semaphore to go to zero
unsigned short   semncnt;   // qty processes waiting for this semaphore to increase value
pid_t            sempid;    // process id that did the last operation
```

- semop() performs operations on the semid set of semaphores.

Semaphores: semop()

int semop(int semid, struct sembuf *sops, unsigned nsops);

- semop() *atomically* performs an array of semaphore operations

  (ie. not-interruptible, as if it were one instruction)

- Each semaphore has the following values associated with it:

```
unsigned short   semval;    // semaphore's value
unsigned short   semzcnt;   // qty processes waiting for this semaphore to go to zero
unsigned short   semncnt;   // qty processes waiting for this semaphore to increase value
pid_t            sempid;    // process id that did the last operation
```

- semop() performs operations on the semid set of semaphores.
- Each of the nsops array members specifies an operation to be performed on a single semaphore.

sembuf covered on next slide

Semaphores: struct sembuf *sops

```
struct sembuf {
        short sem_num;      // semaphore number to be affected
        short sem_op;       // semaphore operation
        short sem_flg;      // semaphore flags
        }
```

- sem_flg may be 0, IPC_NOWAIT, and SEM_UNDO. The latter means: the operation will be automatically undone when the process terminates.

Semaphores: struct sembuf *sops

```
struct sembuf {
        short sem_num;      // semaphore number to be affected
        short sem_op;       // semaphore operation
        short sem_flg;      // semaphore flags
        }
```

- sem_flg may be 0, IPC_NOWAIT, and SEM_UNDO. The latter means: the operation will be automatically undone when the process terminates.
- Operations in sops are performed in array order and atomically

  (done as a complete unit or not at all)

Semaphores: struct sembuf *sops

```
struct sembuf {
        short sem_num;      // semaphore number to be affected
        short sem_op;       // semaphore operation
        short sem_flg;      // semaphore flags
        }
```

- sem_flg may be 0, IPC_NOWAIT, and SEM_UNDO. The latter means: the operation will be automatically undone when the process terminates.
- Operations in sops are performed in array order and atomically

  (done as a complete unit or not at all)
- There are three types of operations available; they depend on the value of sem_op

(*see next slide*)

Semaphores: operations

$sem\_op < 0$ Process must have alter permission on the semaphore set.

Semaphores: operations

$sem\_op < 0$ Process must have alter permission on the semaphore set.

semval $\geq$ |sem_op|: semval $-=$ |sem_op| positive semval $\rightarrow$ zero

Semaphores: operations

$sem\_op < 0$ Process must have alter permission on the semaphore set.

semval $\geq$ |sem_op|: semval $-=$ |sem_op|	positive semval $\rightarrow$ zero
(semval $<$ |sem_op|) $\wedge$ (sem_flg & IPC_NOWAIT)	returns immediately

Semaphores: operations

$sem\_op < 0$ Process must have alter permission on the semaphore set.

semval $\geq$ |sem_op|: semval $-=$ |sem_op|	positive semval $\rightarrow$ zero
(semval $<$ |sem_op|) $\wedge$ (sem_flg & IPC_NOWAIT)	returns immediately
(semval $<$ |sem_op|) $\wedge \sim$ (sem_flg & IPC_NOWAIT)	++semncnt, suspend until semval $\geq$ |semop |

Semaphores: operations

$sem\_op < 0$ Process must have alter permission on the semaphore set.

semval $\geq$ |sem_op|: semval $-=$ |sem_op|	positive semval $\rightarrow$ zero returns immediately
(semval $<$ |sem_op|) $\wedge$ (sem_flg & IPC_NOWAIT)	++semncnt, suspend until semval $\geq$ |semop |
(semval $<$ |sem_op|) $\wedge \sim$ (sem_flg & IPC_NOWAIT)	

$sem\_op > 0$ semval $+=$ sem_op

Semaphores: operations

$sem\_op < 0$ Process must have alter permission on the semaphore set.

semval $\geq$ |sem_op|: semval $-=$ |sem_op|

(semval $<$ |sem_op|) $\wedge$ (sem_flg & IPC_NOWAIT)

(semval $<$ |sem_op|) $\wedge \sim$ (sem_flg & IPC_NOWAIT)

positive semval $\rightarrow$ zero returns immediately

++semncnt, suspend until semval $\geq$ |semop |

$sem\_op > 0$ semval $+=$ sem_op

$sem\_op \equiv 0$

Semaphores: operations

$sem\_op < 0$ Process must have alter permission on the semaphore set.

semval $\geq$ |sem_op|: semval $-=$ |sem_op|	positive semval $\rightarrow$ zero
(semval $<$ |sem_op|) $\wedge$ (sem_flg & IPC_NOWAIT)	returns immediately
(semval $<$ |sem_op|) $\wedge \sim$ (sem_flg & IPC_NOWAIT)	++semncnt, suspend until semval $\geq$ |semop |

$sem\_op > 0$ semval $+=$ sem_op

$sem\_op \equiv 0$

semval $\equiv 0$	return

Semaphores: operations

$sem\_op < 0$ Process must have alter permission on the semaphore set.

semval $\geq$ |sem_op|: semval $-=$ |sem_op|	positive semval $\rightarrow$ zero
(semval $<$ |sem_op|) $\wedge$ (sem_flg & IPC_NOWAIT)	returns immediately
(semval $<$ |sem_op|) $\wedge \sim$ (sem_flg & IPC_NOWAIT)	++semncnt, suspend until semval $\geq$ |semop |

$sem\_op > 0$ semval $+=$ sem_op

$sem\_op \equiv 0$

semval $\equiv 0$	return
semval $\neq 0 \wedge$ (sem_flg & IPC_NOWAIT)	return immediately

Semaphores: operations

$sem\_op < 0$ Process must have alter permission on the semaphore set.

semval $\geq$ |sem_op|: semval $-=$ |sem_op|	positive semval $\rightarrow$ zero
(semval $<$ |sem_op|) $\wedge$ (sem_flg & IPC_NOWAIT)	returns immediately
(semval $<$ |sem_op|) $\wedge \sim$ (sem_flg & IPC_NOWAIT)	++semncnt, suspend until semval $\geq$ |semop |

$sem\_op > 0$ semval $+=$ sem_op

$sem\_op \equiv 0$

semval $\equiv 0$	return
semval $\neq 0 \wedge$ (sem_flg & IPC_NOWAIT)	return immediately
semval $\neq 0 \wedge \sim$ (sem_flg & IPC_NOWAIT)	++semzcnt, suspend until semval becomes zero, semid is removed, $--$semzcnt, calling process receives a signal

(recall: semzcnt holds the qty processes waiting for the semaphore to go to zero)

Semaphores: An Example

- Consider the use of semaphores to control access to shared memory.

Semaphores: An Example

- Consider the use of semaphores to control access to shared memory.
  Interrupts which change process' execution state can occur at almost any point.

Semaphores: An Example

- Consider the use of semaphores to control access to shared memory.
  Interrupts which change process' execution state can occur at almost any point.
  Assume we have two processes:

Semaphores: An Example

- Consider the use of semaphores to control access to shared memory.
  Interrupts which change process' execution state can occur at almost any point.
  Assume we have two processes:
  - **S** sockets -oriented process: typically executes commands very quickly and often, but...

Semaphores: An Example

- Consider the use of semaphores to control access to shared memory.
  Interrupts which change process' execution state can occur at almost any point.
  Assume we have two processes:

  S sockets -oriented process: typically executes commands very quickly and often, but...

  G graphics-oriented process: G's accesses are limited by graphics-refresh cycle considerations.

Semaphores: An Example

- Consider the use of semaphores to control access to shared memory.
  Interrupts which change process' execution state can occur at almost any point.
  Assume we have two processes:

  S sockets -oriented process: typically executes commands very quickly and often, but...

  G graphics-oriented process: G's accesses are limited by graphics-refresh cycle considerations.

  G displays a robot with joint angles $\theta_1, \theta_2, \ldots, \theta_6$ and so needs snapshots of joint angles.

Semaphores: An Example

- Consider the use of semaphores to control access to shared memory.
  Interrupts which change process' execution state can occur at almost any point.
  Assume we have two processes:

  S sockets -oriented process: typically executes commands very quickly and often, but...

  G graphics-oriented process: G's accesses are limited by graphics-refresh cycle considerations.

  G displays a robot with joint angles $\theta_1, \theta_2, \ldots, \theta_6$ and so needs snapshots of joint angles.
  S may change $(\theta_1, \theta_2, \ldots, \theta_6)$ frequently in a control cycle (> 200Hz)

Semaphores: An Example

- Consider the use of semaphores to control access to shared memory.
  Interrupts which change process' execution state can occur at almost any point.
  Assume we have two processes:

  S sockets -oriented process: typically executes commands very quickly and often, but...

  G graphics-oriented process: G's accesses are limited by graphics-refresh cycle considerations.

  G displays a robot with joint angles $\theta_1, \theta_2, \ldots, \theta_6$ and so needs snapshots of joint angles.
  S may change $(\theta_1, \theta_2, \ldots, \theta_6)$ frequently in a control cycle (> 200Hz)

- Assume we work with shared memory

```
struct RobotPosn_str {
  int state;
  float joint[6];
} rp;
```

Semaphores: An Example

- Consider the use of semaphores to control access to shared memory.
  Interrupts which change process' execution state can occur at almost any point.
  Assume we have two processes:

  S sockets -oriented process: typically executes commands very quickly and often, but...

  G graphics-oriented process: G's accesses are limited by graphics-refresh cycle considerations.

  G displays a robot with joint angles $\theta_1, \theta_2, \ldots, \theta_6$ and so needs snapshots of joint angles.
  S may change $(\theta_1, \theta_2, \ldots, \theta_6)$ frequently in a control cycle (> 200Hz)

- Assume we work with shared memory

```
struct RobotPosn_str {
  int state;
  float joint[6];
} rp;
```

- and let `#define DIRTY 1` and `#define CLEAN 0`

Semaphores: An Example

- Assume that `RobotPosn_str rp` lies in shared memory, so both S and G processes can access `rp`.

Semaphores: An Example

- Assume that `RobotPosn_str rp` lies in shared memory, so both S and G processes can access `rp`.
- We don't want S to halfway load `rp.join[]` when G starts reading it; we want *all or nothing*!

Semaphores: An Example

- Assume that `RobotPosn_str rp` lies in shared memory, so both S and G processes can access `rp`.
- We don't want S to halfway load `rp.join[]` when G starts reading it; we want *all or nothing*!
- Semaphores allow us to do this

Semaphores: An Example

- Assume that RobotPosn_str rp lies in shared memory, so both S and G processes can access rp.
- We don't want S to halfway load rp.join[] when G starts reading it; we want *all or nothing*!
- Semaphores allow us to do this
  - G: creates a semaphore: semid= semget(0x123f,1,IPC_CREATE|IPC_EXCL)

Semaphores: An Example

- Assume that RobotPosn_str rp lies in shared memory, so both S and G processes can access rp.
- We don't want S to halfway load rp.join[] when G starts reading it; we want *all or nothing*!
- Semaphores allow us to do this
  - G: creates a semaphore: semid= semget(0x123f,1,IPC_CREATE|IPC_EXCL)
  - G: initializes semaphore value to 1, which will mean "unlocked" or "available"

    semctl(semid,0,SETVAL,1); (which sets semaphore#0 to 1)

Semaphores: An Example

- Assume that RobotPosn_str rp lies in shared memory, so both S and G processes can access rp.
- We don't want S to halfway load rp.join[] when G starts reading it; we want *all or nothing*!
- Semaphores allow us to do this
  - G: creates a semaphore: semid= semget(0x123f,1,IPC_CREATE|IPC_EXCL)
  - G: initializes semaphore value to 1, which will mean "unlocked" or "available"

    semctl(semid,0,SETVAL,1); (which sets semaphore#0 to 1)
- G then uses fork() and exec() to kick off the S process.

Semaphores: An Example

- Assume that RobotPosn_str rp lies in shared memory, so both S and G processes can access rp.
- We don't want S to halfway load rp.join[] when G starts reading it; we want *all or nothing*!
- Semaphores allow us to do this
  - G: creates a semaphore: semid= semget(0x123f,1,IPC_CREATE|IPC_EXCL)
  - G: initializes semaphore value to 1, which will mean "unlocked" or "available"

    semctl(semid,0,SETVAL,1); (which sets semaphore#0 to 1)
- G then uses fork() and exec() to kick off the S process.
  - S: semid= semget(0x123f,IPC_CREAT); (S gets access to the semaphore)

Semaphores: An Example

- Assume that RobotPosn_str rp lies in shared memory, so both S and G processes can access rp.
- We don't want S to halfway load rp.join[] when G starts reading it; we want *all or nothing*!
- Semaphores allow us to do this
  - G: creates a semaphore: semid= semget(0x123f,1,IPC_CREATE|IPC_EXCL)
  - G: initializes semaphore value to 1, which will mean "unlocked" or "available"

    semctl(semid,0,SETVAL,1); (which sets semaphore#0 to 1)
- G then uses fork() and exec() to kick off the S process.
  - S: semid= semget(0x123f,IPC_CREAT); (S gets access to the semaphore)
- Assume S gets a message from a socket to set the robot's joint angles.

Semaphores: An Example

- S then updates the `RobotPosn rp` data structure:

Semaphores: An Example

- S then updates the RobotPosn rp data structure:
  - S: sops.sem_num= 0 (semaphore#0)

Semaphores: An Example

- S then updates the RobotPosn rp data structure:
  - S: sops.sem_num= 0 (semaphore#0)
  - S: sops.sem_op= -1 (semop< 0: so semval is decremented)

Semaphores: An Example

- S then updates the RobotPosn rp data structure:
  - S: sops.sem_num= 0 (semaphore#0)
  - S: sops.sem_op= -1 (semop< 0: so semval is decremented)
  - S: sops.sem_flg = 0 (block if unavailable)

Semaphores: An Example

- S then updates the RobotPosn rp data structure:
  - S: sops.sem_num= 0 (semaphore#0)
  - S: sops.sem_op= -1 (semop$<$ 0: so semval is decremented)
  - S: sops.sem_flg = 0 (block if unavailable)
  - S: semop(semid,&sops,(size_t)1); (acquire resource when semval $\rightarrow$ 0)

Semaphores: An Example

- S then updates the RobotPosn rp data structure:
  - S: sops.sem_num= 0 (semaphore#0)
  - S: sops.sem_op= -1 (semop$<$ 0: so semval is decremented)
  - S: sops.sem_flg = 0 (block if unavailable)
  - S: semop(semid,&sops,(size_t)1); (acquire resource when semval $\rightarrow$ 0)
  - S: rp.state= DIRTY (indicate that resource has been chgd)

Semaphores: An Example

- S then updates the RobotPosn rp data structure:
  - S: sops.sem_num= 0 (semaphore#0)
  - S: sops.sem_op= -1 (semop$<$ 0: so semval is decremented)
  - S: sops.sem_flg = 0 (block if unavailable)
  - S: semop(semid,&sops,(size_t)1); (acquire resource when semval $\rightarrow$ 0)
  - S: rp.state= DIRTY (indicate that resource has been chgd)
  - S: rp.joint[0]=$\cdots$ (changing resource)

Semaphores: An Example

- S then updates the RobotPosn rp data structure:
  - S: sops.sem_num= 0 (semaphore#0)
  - S: sops.sem_op= -1 (semop$<$ 0: so semval is decremented)
  - S: sops.sem_flg = 0 (block if unavailable)
  - S: semop(semid,&sops,(size_t)1); (acquire resource when semval $\rightarrow$ 0)
  - S: rp.state= DIRTY (indicate that resource has been chgd)
  - S: rp.joint[0]=$\cdots$ (changing resource)
  - S: sops.sem_op= 1; (semop $>$ 0: semval is incremented)

Semaphores: An Example

- S then updates the RobotPosn rp data structure:
  - S: sops.sem_num= 0 (semaphore#0)
  - S: sops.sem_op= -1 (semop$<$ 0: so semval is decremented)
  - S: sops.sem_flg = 0 (block if unavailable)
  - S: semop(semid,&sops,(size_t)1); (acquire resource when semval $\rightarrow$ 0)
  - S: rp.state= DIRTY (indicate that resource has been chgd)
  - S: rp.joint[0]=$\cdots$ (changing resource)
  - S: sops.sem_op= 1; (semop $>$ 0: semval is incremented)
  - S: semop(semid,&sops,(size_t) 1);(release resource, semval $\rightarrow$ 1)

Semaphores: An Example

- In this small example, we subsequently could have

Semaphores: An Example

- In this small example, we subsequently could have
  - G: sops.sem_num= 0 (semaphore#0)

Semaphores: An Example

- In this small example, we subsequently could have
  - G: sops.sem_num= 0 (semaphore#0)
  - G: sops.sem_op= -1 (semop< 0: so semval–)

Semaphores: An Example

- In this small example, we subsequently could have
  - G: sops.sem_num= 0 (semaphore#0)
  - G: sops.sem_op= -1 (semop< 0: so semval–)
  - G: sops.sem_flg = 0 (block if unavailable)

Semaphores: An Example

- In this small example, we subsequently could have
  - G: sops.sem_num= 0 (semaphore#0)
  - G: sops.sem_op= -1 (semop$<$ 0: so semval–)
  - G: sops.sem_flg = 0 (block if unavailable)
  - G: semop(semid,&sops,(size_t)1); (acquire resource when semval $\rightarrow$ 0)

Semaphores: An Example

- In this small example, we subsequently could have
  - G: sops.sem_num= 0 (semaphore#0)
  - G: sops.sem_op= -1 (semop$<$ 0: so semval–)
  - G: sops.sem_flg = 0 (block if unavailable)
  - G: semop(semid,&sops,(size_t)1); (acquire resource when semval $\rightarrow$ 0)
  - G: copy rp.joints from shared memory to G's private copy

Semaphores: An Example

- In this small example, we subsequently could have
  - G: sops.sem_num= 0 (semaphore#0)
  - G: sops.sem_op= -1 (semop$<$ 0: so semval–)
  - G: sops.sem_flg = 0 (block if unavailable)
  - G: semop(semid,&sops,(size_t)1); (acquire resource when semval $\rightarrow$ 0)
  - G: copy rp.joints from shared memory to G's private copy
  - G: rp.state= CLEAN (indicate that resource has been chgd)

Semaphores: An Example

- In this small example, we subsequently could have
  - G: sops.sem_num= 0 (semaphore#0)
  - G: sops.sem_op= -1 (semop$<$ 0: so semval–)
  - G: sops.sem_flg = 0 (block if unavailable)
  - G: semop(semid,&sops,(size_t)1); (acquire resource when semval $\rightarrow$ 0)
  - G: copy rp.joints from shared memory to G's private copy
  - G: rp.state= CLEAN (indicate that resource has been chgd)
  - G: sops.sem_op= 1; (semop $>$ 0: semval++)

Semaphores: An Example

- In this small example, we subsequently could have
  - G: sops.sem_num= 0 (semaphore#0)
  - G: sops.sem_op= -1 (semop$<$ 0: so semval–)
  - G: sops.sem_flg = 0 (block if unavailable)
  - G: semop(semid,&sops,(size_t)1); (acquire resource when semval $\rightarrow$ 0)
  - G: copy rp.joints from shared memory to G's private copy
  - G: rp.state= CLEAN (indicate that resource has been chgd)
  - G: sops.sem_op= 1; (semop $>$ 0: semval++)
  - G: semop(semid,&sops,(size_t) 1);(release resource, semval $\rightarrow$ 1)

Semaphores: An Example

- In this small example, we subsequently could have
  - G: sops.sem_num= 0 (semaphore#0)
  - G: sops.sem_op= -1 (semop$<$ 0: so semval–)
  - G: sops.sem_flg = 0 (block if unavailable)
  - G: semop(semid,&sops,(size_t)1); (acquire resource when semval $\rightarrow$ 0)
  - G: copy rp.joints from shared memory to G's private copy
  - G: rp.state= CLEAN (indicate that resource has been chgd)
  - G: sops.sem_op= 1; (semop $>$ 0: semval++)
  - G: semop(semid,&sops,(size_t) 1);(release resource, semval $\rightarrow$ 1)
- In other words, G has nothing better to do but wait until S updates the robot position.

Semaphores: An Example

- In this small example, we subsequently could have
  - G: sops.sem_num= 0 (semaphore#0)
  - G: sops.sem_op= -1 (semop$<$ 0: so semval–)
  - G: sops.sem_flg = 0 (block if unavailable)
  - G: semop(semid,&sops,(size_t)1); (acquire resource when semval $\rightarrow$ 0)
  - G: copy rp.joints from shared memory to G's private copy
  - G: rp.state= CLEAN (indicate that resource has been chgd)
  - G: sops.sem_op= 1; (semop $>$ 0: semval++)
  - G: semop(semid,&sops,(size_t) 1);(release resource, semval $\rightarrow$ 1)
- In other words, G has nothing better to do but wait until S updates the robot position.
- More practically, S may potentially update a number of things, each of which is assigned its own semaphore from the set.

Semaphores: An Example

- In this small example, we subsequently could have
  - G: sops.sem_num= 0 (semaphore#0)
  - G: sops.sem_op= -1 (semop$<$ 0: so semval–)
  - G: sops.sem_flg = 0 (block if unavailable)
  - G: semop(semid,&sops,(size_t)1); (acquire resource when semval $\rightarrow$ 0)
  - G: copy rp.joints from shared memory to G's private copy
  - G: rp.state= CLEAN (indicate that resource has been chgd)
  - G: sops.sem_op= 1; (semop $>$ 0: semval++)
  - G: semop(semid,&sops,(size_t) 1);(release resource, semval $\rightarrow$ 1)
- In other words, G has nothing better to do but wait until S updates the robot position.
- More practically, S may potentially update a number of things, each of which is assigned its own semaphore from the set.
- G may need to pay attention to a mouse or joystick, in which case it may be suspended while awaiting a mouse queue event

Semaphores: An Example

- S could then make a batch of changes to shared memory, and then send G a SIGUSR to wake G up:

Semaphores: An Example

- S could then make a batch of changes to shared memory, and then send G a SIGUSR to wake G up:
  - S: kill(getppid(),SIGUSR1)

Semaphores: An Example

- S could then make a batch of changes to shared memory, and then send G a SIGUSR to wake G up:
  - S: kill(getppid(),SIGUSR1)
- To handle this properly, G should initialize a signal handler:

Semaphores: An Example

- S could then make a batch of changes to shared memory, and then send G a SIGUSR to wake G up:
  - S: kill(getppid(),SIGUSR1)
- To handle this properly, G should initialize a signal handler:
  - G: signal(SIGUSR1,sigusr1_handler);

    (or sigset() or sigaction())

Semaphores: An Example

- S could then make a batch of changes to shared memory, and then send G a SIGUSR to wake G up:
  - S: kill(getppid(),SIGUSR1)
- To handle this properly, G should initialize a signal handler:
  - G: signal(SIGUSR1,sigusr1_handler);

    (or sigset() or sigaction())
  - G: where

```
int sigusr1= 0;
void sigusr1_handler(int sig)
{
        signal(SIGUSR1,sigusr1_handler);
        sigusr1= 1;
}
```

Semaphores: An Example

- S could then make a batch of changes to shared memory, and then send G a SIGUSR to wake G up:
  - S: kill(getppid(),SIGUSR1)
- To handle this properly, G should initialize a signal handler:
  - G: signal(SIGUSR1,sigusr1_handler);

    (or sigset() or sigaction())
  - G: where

```
int sigusr1= 0;
void sigusr1_handler(int sig)
{
        signal(SIGUSR1,sigusr1_handler);
        sigusr1= 1;
}
```

- Receipt of a signal will break a process out of a suspended state.

Semaphores: An Example

- S could then make a batch of changes to shared memory, and then send G a SIGUSR to wake G up:
  - S: kill(getppid(),SIGUSR1)
- To handle this properly, G should initialize a signal handler:
  - G: signal(SIGUSR1,sigusr1_handler);

    (or sigset() or sigaction())
  - G: where

```
int sigusr1= 0;
void sigusr1_handler(int sig)
{
        signal(SIGUSR1,sigusr1_handler);
        sigusr1= 1;
}
```

- Receipt of a signal will break a process out of a suspended state.
- The select() call (see sockets) will return a -1 with errno=EINTR in such a case.