OPERATING SYSTEMS

IPC - Shared Memory & Message Passing

Suresh Jamadagni
Department of Computer Science

OPERATING SYSTEMS

Slides Credits for all the PPTs of this course

- The slides/diagrams in this course are an **adaptation**, **combination**, and **enhancement** of material from the following resources and persons:

1. Slides of Operating System Concepts, Abraham Silberschatz, Peter Baer Galvin, Greg Gagne - 9th edition 2013 and some slides from 10th edition 2018
2. Some conceptual text and diagram from Operating Systems - Internals and Design Principles, William Stallings, 9th edition 2018
3. Some presentation transcripts from A. Frank – P. Weisberg
4. Some conceptual text from Operating Systems: Three Easy Pieces, Remzi Arpaci-Dusseau, Andrea Arpaci Dusseau

OPERATING SYSTEMS

IPC - Shared Memory and Message Passing

Suresh Jamadagni
Department of Computer Science

OPERATING SYSTEMS
Interprocess Communication

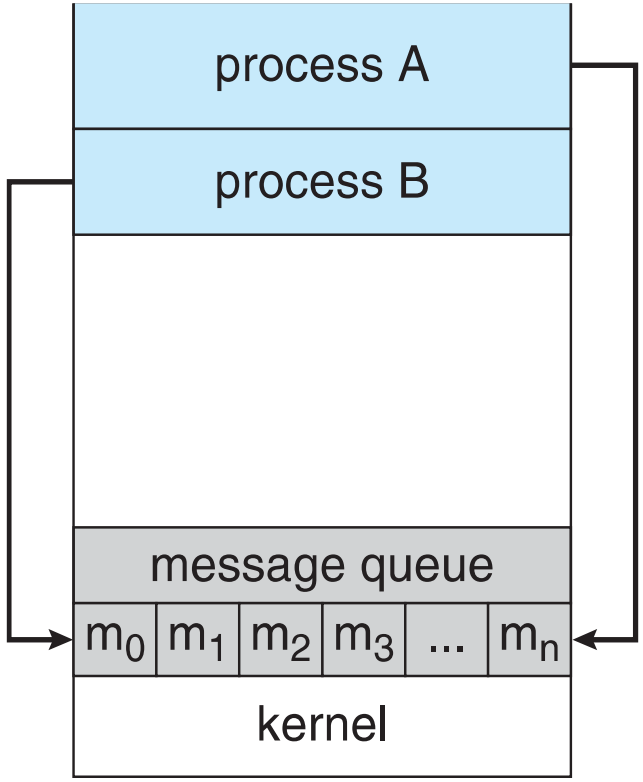
- Processes within a system may be ***independent*** or ***cooperating***
- Cooperating process can affect or be affected by other processes, including sharing data
- Reasons for cooperating processes:
  - Information sharing
  - Computation speedup
  - Modularity
  - Convenience
- Cooperating processes need **interprocess communication** (**IPC**)
- Two models of IPC
  - **Shared memory**
  - **Message passing**

Cooperating Processes

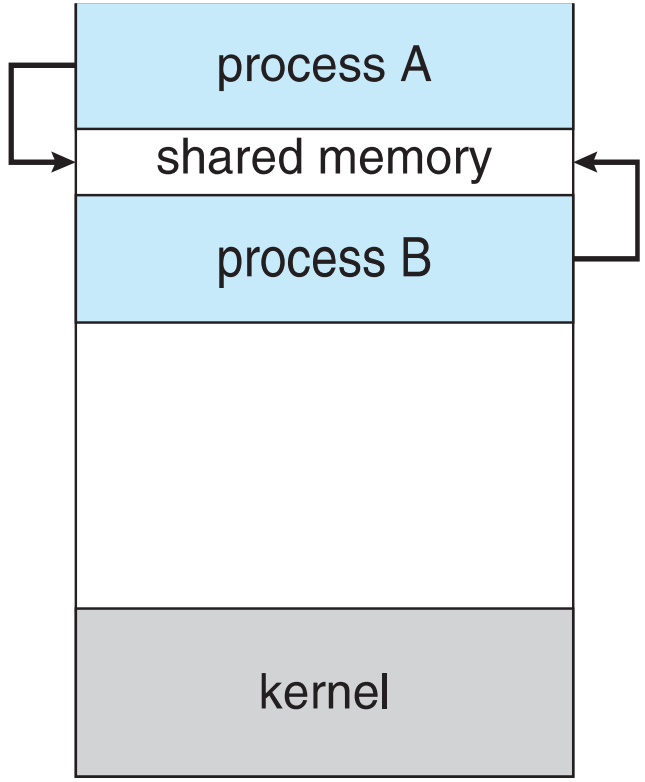
- ***Independent*** process cannot affect or be affected by the execution of another process
- ***Cooperating*** process can affect or be affected by the execution of another process
- Advantages of process cooperation
  - Information sharing
  - Computation speed-up
  - Modularity
  - Convenience

Communication Models

- Two models of IPC
  a) **Message passing**
  b) **Shared memory**



(a)



(b)

Producer-Consumer Problem

- Paradigm for cooperating processes, *producer* process produces information that is consumed by a *consumer* process
  - **unbounded-buffer** places no practical limit on the size of the buffer
    - Consumer may have to wait for new items, but the producer can always produce new items
  - **bounded-buffer** assumes that there is a fixed buffer size
    - Consumer must wait if the buffer is empty; producer must wait if the buffer is full

OPERATING SYSTEMS

Bounded-Buffer – Producer Consumer

- Shared data

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

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

- Shared buffer is implemented as a circular array with 2 logical pointers: **in** and **out**
- Buffer is empty when **in == out;** buffer is full when ((**in** + 1) % BUFFER_SIZE) == **out**
- Variable **in** points to the next free position in the buffer; **out** points to the first full position in the buffer
- Solution is correct, but can only use BUFFER_SIZE-1 elements

Bounded-Buffer – Producer

```
item next_produced;
while (true) {
        /* produce an item in next_produced */
        while (((in + 1) % BUFFER_SIZE) == out)
                ; /* do nothing */
        buffer[in] = next_produced;
        in = (in + 1) % BUFFER_SIZE;
}
```

Bounded-Buffer – Consumer

```
item next_consumed;

while (true) {
        while (in == out)
                ; /* do nothing */
        next_consumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;

        /* consume the item in next_consumed */
}
```

Interprocess Communication – Shared Memory

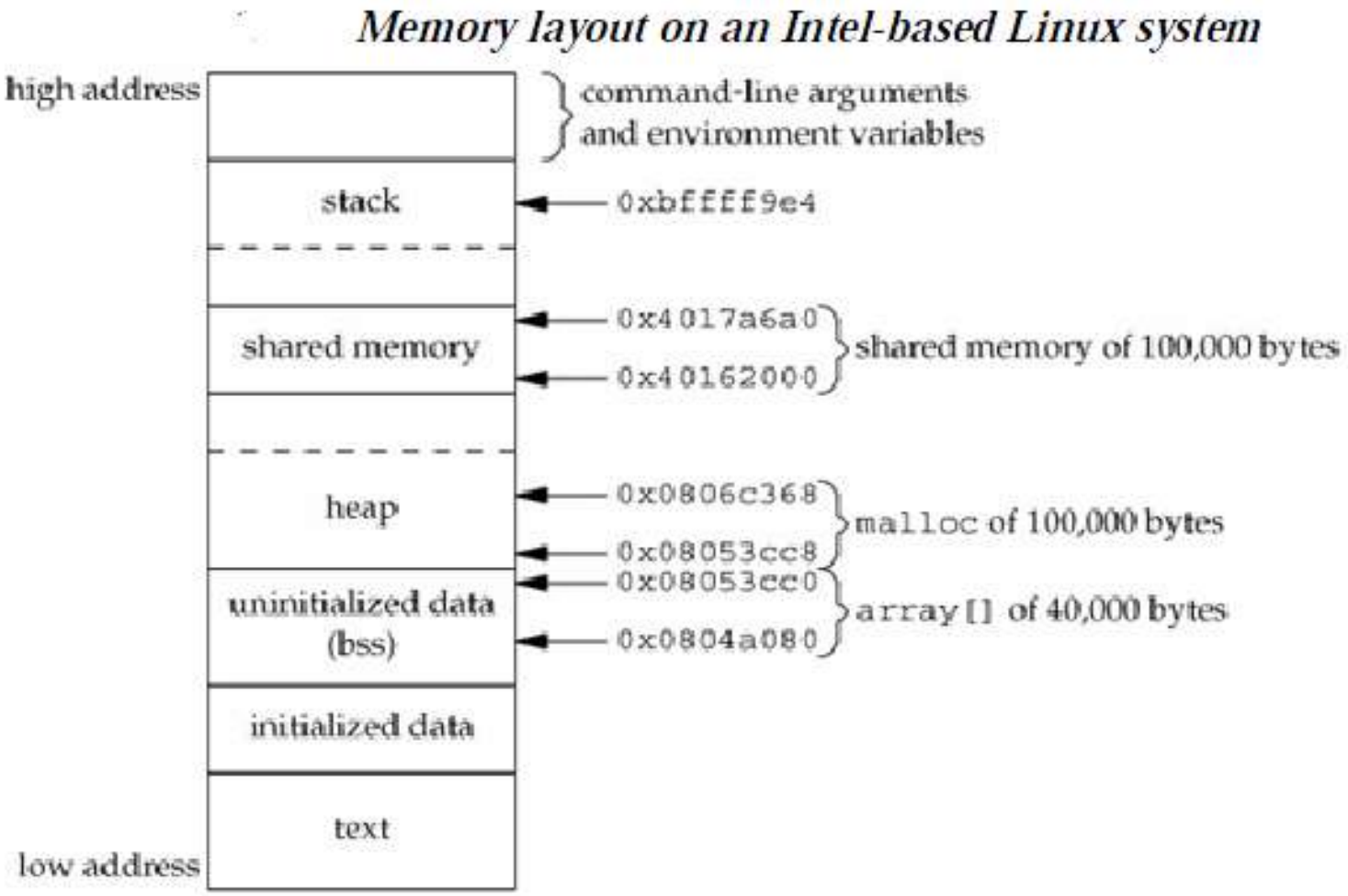
- An area of memory shared among the processes that wish to communicate
- The communication is under the control of the users processes not the operating system.
- Major issues is to provide mechanism that will allow the user processes to synchronize their actions when they access shared memory.

Shared Memory (Cont.)

- Shared memory allows two or more processes to share a given region of memory.
- Shared memory is the fastest form of IPC, because the data does not need to be copied between the client and the server.
- The only trick in using shared memory is synchronizing access to a given region among multiple processes.
- If the server is placing data into a shared memory region, the client shouldn't try to access the data until the server is done.
- Often, semaphores are used to synchronize shared memory access. (record locking can also be used.)

OPERATING SYSTEMS

Shared Memory (Cont.)



Interprocess Communication – Message Passing

- Mechanism for processes to communicate and to synchronize their actions
- Message system – processes communicate with each other without resorting to shared variables
- IPC facility provides two operations:
  - **send**(*message*)
  - **receive**(*message*)
- The *message* size is either fixed or variable

OPERATING SYSTEMS

Message Passing (Cont.)

- If processes *P* and *Q* wish to communicate, they need to:
  - Establish a ***communication link*** between them
  - Exchange messages via send/receive
- Implementation issues:
  - How are links established?
  - Can a link be associated with more than two processes?
  - How many links can there be between every pair of communicating processes?
  - What is the capacity of a link?
  - Is the size of a message that the link can accommodate fixed or variable?
  - Is a link unidirectional or bi-directional?

Message Passing (Cont.)

- Implementation of communication link
  - Physical:
    - Shared memory
    - Hardware bus
    - Network
  - Logical:
    - Direct or indirect
    - Synchronous or asynchronous
    - Automatic or explicit buffering

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
- Properties of communication link
  - Links are established automatically
  - A link is associated with exactly one pair of communicating processes
  - Between each pair there exists exactly one link
  - The link may be unidirectional, but is usually bi-directional

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
- Properties of communication link
  - Link established only if processes share a common mailbox
  - A link may be associated with many processes
  - Each pair of processes may share several communication links
  - Link may be unidirectional or bi-directional

OPERATING SYSTEMS

Indirect Communication (Cont.)

- Operations
  - create a new mailbox (port)
  - send and receive messages through mailbox
  - destroy a mailbox
- Primitives are defined as:
- **send**(*A, message*) – send a message to mailbox A
- **receive**(*A, message*) – receive a message from mailbox A

Indirect Communication (Cont.)

- Mailbox sharing
  - P_1, P_2, and P_3 share mailbox A
  - P_1 sends; P_2 and P_3 receive
  - Who gets the message?
- Solutions
  - Allow a link to be associated with at most two processes
  - Allow only one process at a time to execute a receive operation
  - Allow the system to select arbitrarily the receiver. Sender is notified who the receiver was.

OPERATING SYSTEMS

Message Passing - Synchronization

- Message passing may be either blocking or non-blocking
- **Blocking** is considered **synchronous**
  - **Blocking send** -- the sender is blocked until the message is received
  - **Blocking receive** -- the receiver is blocked until a message is available
- **Non-blocking** is considered **asynchronous**
  - **Non-blocking send** -- the sender sends the message and continue
  - **Non-blocking receive** -- the receiver receives:
    - A valid message, or
    - Null message
- Different combinations possible
  - If both send and receive are blocking, we have a **rendezvous** between the sender and the receiver

Message Passing - Synchronization (Cont.)

- Producer-consumer becomes trivial

```
 message next_produced;
while (true) {
   /* produce an item in next_produced */
send(next_produced);
}
```

```
message next_consumed;
while (true) {
  receive(next_consumed);

  /* consume the item in next_consumed */
}
```

OPERATING SYSTEMS

Buffering

- Queue of messages attached to the link (direct or indirect); messages reside in a temporary queue.
- Queues can be implemented in one of three ways
    1. Zero capacity – no messages are queued on a link.

       Sender must wait for receiver (rendezvous)

    2. Bounded capacity – finite length of *n* messages

       Sender must wait if link full

    3. Unbounded capacity – infinite length

       Sender never waits
- Zero-capacity case is sometimes referred to as a message system with no buffering; other cases are referred to as systems with automatic buffering

THANK YOU

Suresh Jamadagni
Department of Computer Science Engineering

sureshjamadagni@pes.edu

UE20CS254

Operating Systems

Suresh Jamadagni

Department of Computer Science and Engineering

Operating Systems

System calls for Inter Process Communication

Suresh Jamadagni
Department of Computer Science and Engineering

Pipes

- Pipes are the oldest form of UNIX System IPC and are provided by all UNIX systems.
- Pipes have two limitations.
    1. Historically, they have been half duplex (i.e., data flows in only one direction). Some systems now provide full-duplex pipes, but for maximum portability, we should never assume that this is the case.
    2. Pipes can be used only between processes that have a **common ancestor**. Normally, a pipe is created by a process, that process calls fork, and the pipe is used between the parent and the child.

pipe()

- A pipe is created by calling the **pipe()** function.

```
#include <unistd.h>
int pipe(int fd[2]);
```

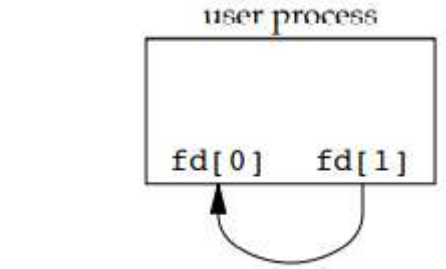
- Return value: 0 if OK, −1 on error
- Two file descriptors are returned through the fd argument:
  1. fd[0] is open for reading
  2. fd[1] is open for writing.
- The output of fd[1] is the input for fd[0]

OPERATING SYSTEMS

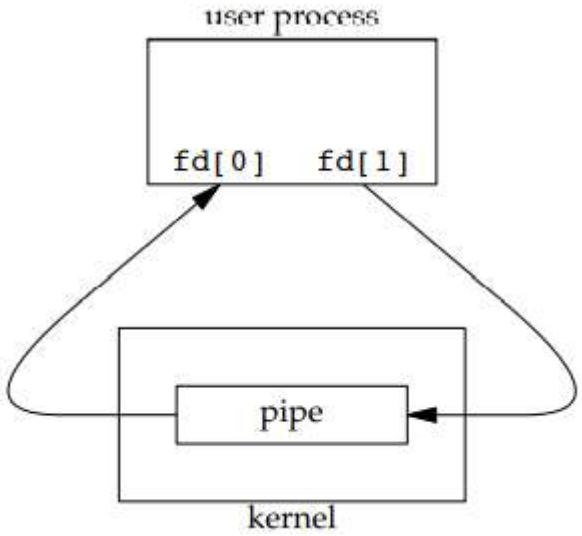
pipe()

- Two ways to picture a half-duplex pipe

1. The two ends of the pipe connected in a single process



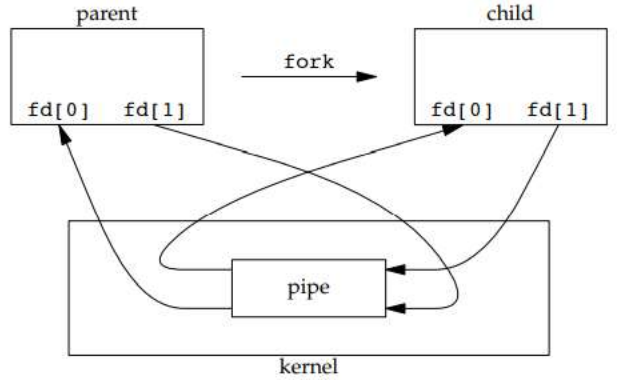
2. The data in the pipe flows through the kernel



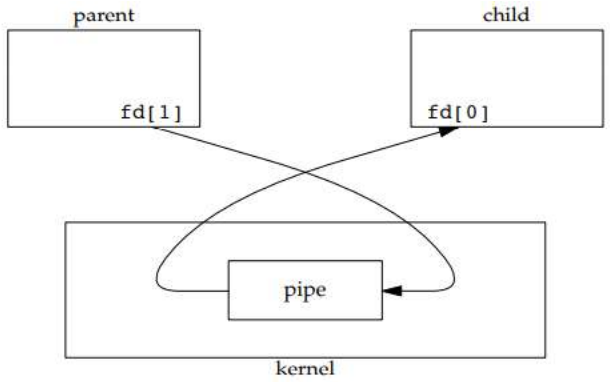
OPERATING SYSTEMS

pipe()

- Normally, a process that calls pipe then calls fork, creating an IPC channel from the parent to the child or vice versa



- For a pipe from the parent to the child, the parent closes the read end of the pipe (fd[0]) and the child closes the write end (fd[1]).
- For a pipe from the child to the parent, the parent closes fd[1], and the child closes fd[0]



Pipe from parent to child

OPERATING SYSTEMS

popen() and pclose()

- A common operation is to create a pipe to another process to either read its output or send it input, the standard I/O library has historically provided the **popen()** and **pclose()** functions
- These two functions handle all the work: creating a pipe, forking a child, closing the unused ends of the pipe, executing a shell to run the command, and waiting for the command to terminate.

```
#include <stdio.h>

FILE *popen(const char *cmdstring, const char *type);


int pclose(FILE *fp);
```

- The function **popen()** does a fork and exec to execute the cmdstring and returns a standard I/O file pointer.
- If type is "r", the file pointer is connected to the standard output of cmdstring.
- If type is "w", the file pointer is connected to the standard input of cmdstring
- The function popen() returns file pointer if OK, NULL on error
- The function **pclose()** closes the standard I/O stream, waits for the command to terminate, and returns the termination status of the shell or -1 on error

OPERATING SYSTEMS

FIFOs

- FIFOs are sometimes called **named pipes**.
- Unnamed pipes can be used only between related processes when a common ancestor has created the pipe.
- With FIFOs, unrelated processes can exchange data
- Creating a FIFO is similar to creating a file

```
#include <sys/stat.h>
int mkfifo(const char *path, mode_t mode);
int mkfifoat(int fd, const char *path, mode_t mode);
```

- Functions mkfifo() and mkfifoat() return 0 if OK, −1 on error
- Once a FIFO has been created using mkfifo() or mkfifoat() , it can be opened using open(). Normal file I/O functions (e.g., close, read, write, unlink) all work with FIFOs.
- There are two uses for FIFOs.
    1. FIFOs are used by shell commands to pass data from one shell pipeline to another without creating intermediate temporary files.
    2. FIFOs are used as rendezvous points in client–server applications to pass data between the clients and the servers

OPERATING SYSTEMS

Message Queues

- A message queue is a linked list of messages stored within the kernel and identified by a message queue identifier
- A new queue is created or an existing queue opened by **msgget**.

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

- Returns message queue ID if OK, −1 on error
- New messages are added to the end of a queue by **msgsnd**.

```
#include <sys/msg.h>
int msgsnd(int msqid, const void *ptr, size_t nbytes, int flag);
```

- Every message has a positive long integer type field, a non-negative length, and the actual data bytes all of which are specified to **msgsnd** when the message is added to a queue.
- Messages are fetched from a queue by **msgrcv**.

```
#include <sys/msg.h>
ssize_t msgrcv(int msqid, void *ptr, size_t nbytes, long type, int flag);
```

- Messages need not be fetched in a first-in, first-out order. Instead, can be fetched based on their type field

OPERATING SYSTEMS

Semaphores

- A semaphore is a counter used to provide access to a shared data object for multiple processes.
- To obtain a shared resource, a process needs to do the following:
  1. Test the semaphore that controls the resource.
  2. If the value of the semaphore is positive, the process can use the resource. In this case, the process decrements the semaphore value by 1, indicating that it has used one unit of the resource.
  3. Otherwise, if the value of the semaphore is 0, the process goes to sleep until the semaphore value is greater than 0. When the process wakes up, it returns to step 1.
- When a process is done with a shared resource that is controlled by a semaphore, the semaphore value is incremented by 1.
- If any other processes are asleep, waiting for the semaphore, they are awakened.

OPERATING SYSTEMS

Semaphores

- First need to obtain a semaphore ID by calling the semget function

```
#include <sys/sem.h>
int semget(key_t key, int nsems, int flag);
```

- Returns: semaphore ID if OK, −1 on error
- **semctl** function is the catchall for various semaphore operations.

```
#include <sys/sem.h>
int semctl(int semid, int semnum, int cmd, ... /* union semun arg */ );
```

- The function **semop** atomically performs an array of operations on a semaphore set

```
#include <sys/sem.h>
int semop(int semid, struct sembuf semoparray[], size_t nops);
```

- The semoparray argument is a pointer to an array of semaphore operations
- Returns 0 if OK, −1 on error

OPERATING SYSTEMS

Shared memory

- Shared memory allows two or more processes to share a given region of memory.
- This is the fastest form of IPC, because the data does not need to be copied between the client and the server
- The challenge in using shared memory is synchronizing access to a given region among multiple processes.
- Semaphores and mutexes are used to synchronize shared memory access
- Function **shmget** is used to obtain a shared memory identifier

```
#include <sys/shm.h>
int shmget(key_t key, size_t size, int flag);
```

- Returns shared memory ID if OK, −1 on error

OPERATING SYSTEMS

Shared memory

- Function **shmctl** is the catchall for various shared memory operations

```
#include <sys/shm.h>
int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

- Returns 0 if OK, −1 on error
- The cmd argument specifies one of the commands to be performed on the segment specified by shmid
- Once a shared memory segment has been created, a process is attached to this memory segment by calling **shmat**

```
#include <sys/shm.h>
void *shmat(int shmid, const void *addr, int flag);
```

- Returns: pointer to shared memory segment if OK, −1 on error

OPERATING SYSTEMS

Shared memory

- Function call **shmdt** will detach a shared memory segment from a process

```
#include <sys/shm.h>
int shmdt(const void *addr);
```

- Returns 0 if OK, −1 on error
- A shared memory segment can be removed by calling **shmctl** with a command of IPC_RMID.

THANK YOU

Suresh Jamadagni
Department of Computer Science and Engineering

sureshjamadagni@pes.edu

OPERATING SYSTEMS

Threads and Concurrency

Suresh Jamadagni
Department of Computer Science

OPERATING SYSTEMS

Slides Credits for all the PPTs of this course

- The slides/diagrams in this course are an **adaptation**, **combination**, and **enhancement** of material from the following resources and persons:

1. Slides of Operating System Concepts, Abraham Silberschatz, Peter Baer Galvin, Greg Gagne - 9th edition 2013 and some slides from 10th edition 2018
2. Some conceptual text and diagram from Operating Systems - Internals and Design Principles, William Stallings, 9th edition 2018
3. Some presentation transcripts from A. Frank – P. Weisberg
4. Some conceptual text from Operating Systems: Three Easy Pieces, Remzi Arpaci-Dusseau, Andrea Arpaci Dusseau

OPERATING SYSTEMS

Threads and Concurrency

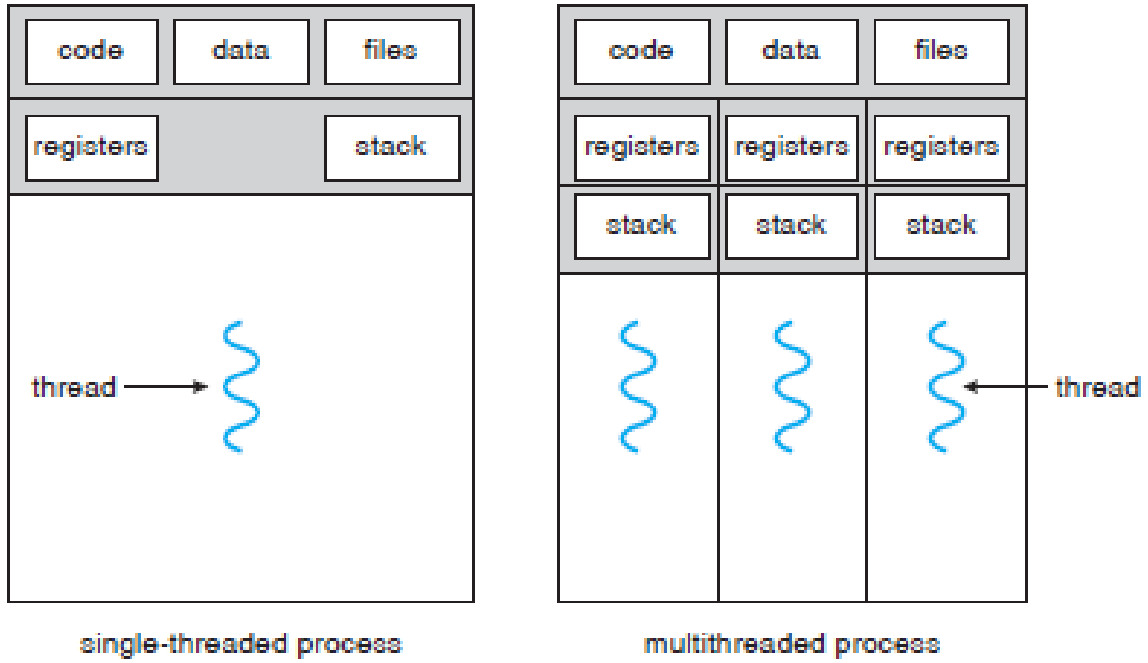
Suresh Jamadagni
Department of Computer Science

Overview and Motivation

- A Thread is a fundamental unit of CPU utilization that forms the basis of multithreaded computer systems.
- It consists of thread ID, Program counter, a register set and stack
- Shares with other threads of same process its code, data,file descriptors, signals
- Most modern applications are multithreaded
- Threads run within application
- Multiple tasks with the application can be implemented by separate threads.
- Application 1: internet browser.
  - numerous tabs open at a given time
  - Multiple threads of execution are used to load content, display animations, play a video, fetch data from a network and so on.

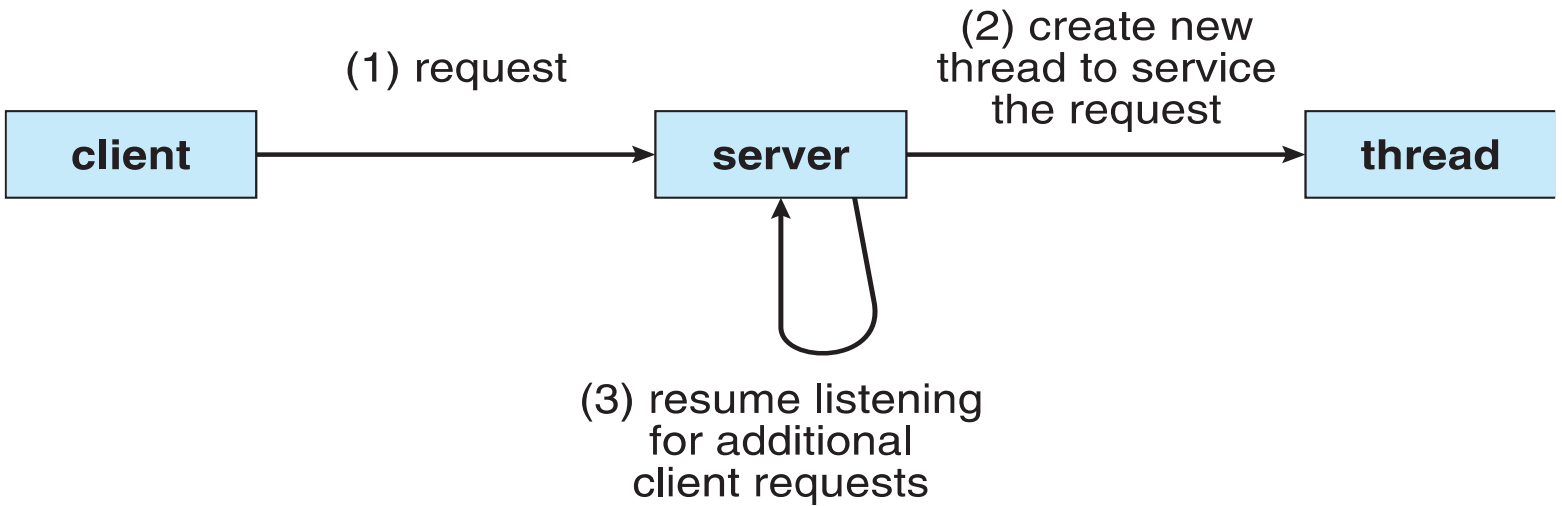
Single threaded and multithreaded processes

- Application 2: Word processor
  - Thread to accept input
  - A thread for spell checking
  - A thread for grammar checking



Multithreaded Server Architecture

- Process creation is heavy-weight while thread creation is light-weight
- Process creation if time consuming, resource intensive
- Threads also play a vital role in remote procedure call (RPC) systems
- Can simplify code, increase efficiency
- Kernels are generally multithreaded



Benefits

- **Responsiveness –** may allow continued execution if part of process is blocked or performing lengthy operations.
  - It is useful in designing user interfaces.
  - A multithreaded Web browser could allow user interaction in one thread while an image was being loaded in another thread.
  - Example: click on the button
- **Resource Sharing –** threads share resources of process, easier than shared memory or message passing.
  - Programmer needs to specify the techniques for sharing
  - But threads share the memory and other resources
  - Sharing of resources helps in having many threads within the same address space

Benefits

- **Economy –** cheaper than process creation, thread switching lower overhead than context switching.
  - In Solaris, for example, creating a process is about thirty times slower than creating a thread, and context switching is about five times slower.
- **Scalability –** process can take advantage of multiprocessor architectures.
  - Threads can run on multiple cores parallelly

OPERATING SYSTEMS

Processes vs Threads

Process

- Will by default not share memory
- Most file descriptors not shared
- Don't share filesystem context
- Don't share signal handling

Thread

- Will by default share memory
- Will share file descriptors
- Will share filesystem context
- Will share signal handling

OPERATING SYSTEMS

Attributes shared by Threads

- process ID and parent process ID;process group ID and session ID;
- controlling terminal;
- process credentials (user and group Ids);open file descriptors;
- record locks created using fcntl();signal dispositions;
- file system–related information: umask, cwd and root directory;
- resource limits;CPU time consumed (as returned by times());
- resources consumed (as returned by getrusage()); nice value (set by setpriority() and nice()).

OPERATING SYSTEMS

Attributes specific to Threads

- thread ID ;signal mask;
- Thread-specific data ;
- the errno variable;
- floating-point environment (see fenv(3));
- stack (local variables and function call linkage information i.e CPU registers saved in the called function's stack frame when one function calls another function and restored for the calling function when the called function returns)
- and a few more …….

OPERATING SYSTEMS

Thread Concepts

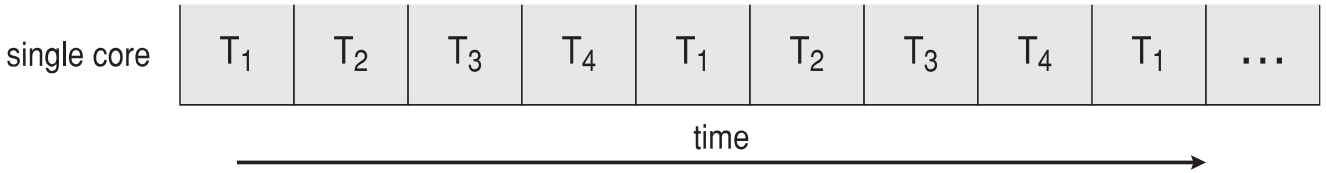
- A thread consists of the following information necessary to represent an execution context within a process.
- thread ID that identifies the thread within a process
- Every thread has a thread ID
- set of register values
- stack
- scheduling priority and policy
- signal mask
- Errno variable
- Thread-specific/thread-private data (each thread to access its own separate copy of the data, without worrying about synchronizing access with other threads)

Thread Concepts (Cont.)

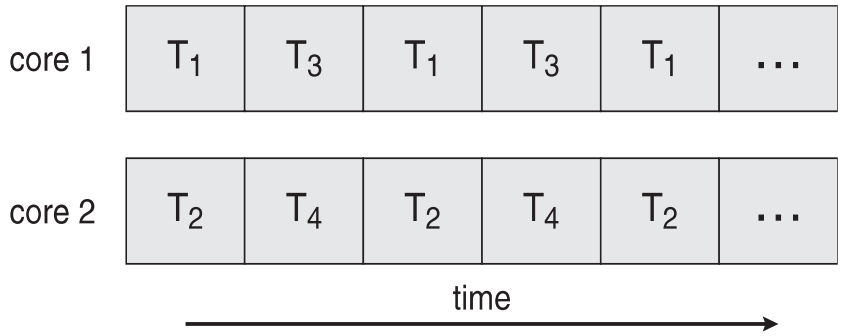
- When a thread is created, there is no guarantee which will run first: the newly created thread or the calling thread.
- The newly created thread has access to the process address space and inherits the calling thread's floating-point environment and signal mask
- The set of pending signals for the thread is cleared.
- The pthread functions usually return an error code when they fail. They don't set errno like the other POSIX functions.

Concurrency vs. Parallelism

- **Concurrent execution on single-core system:**



- **Parallelism on a multi-core system:**



Multicore Programming

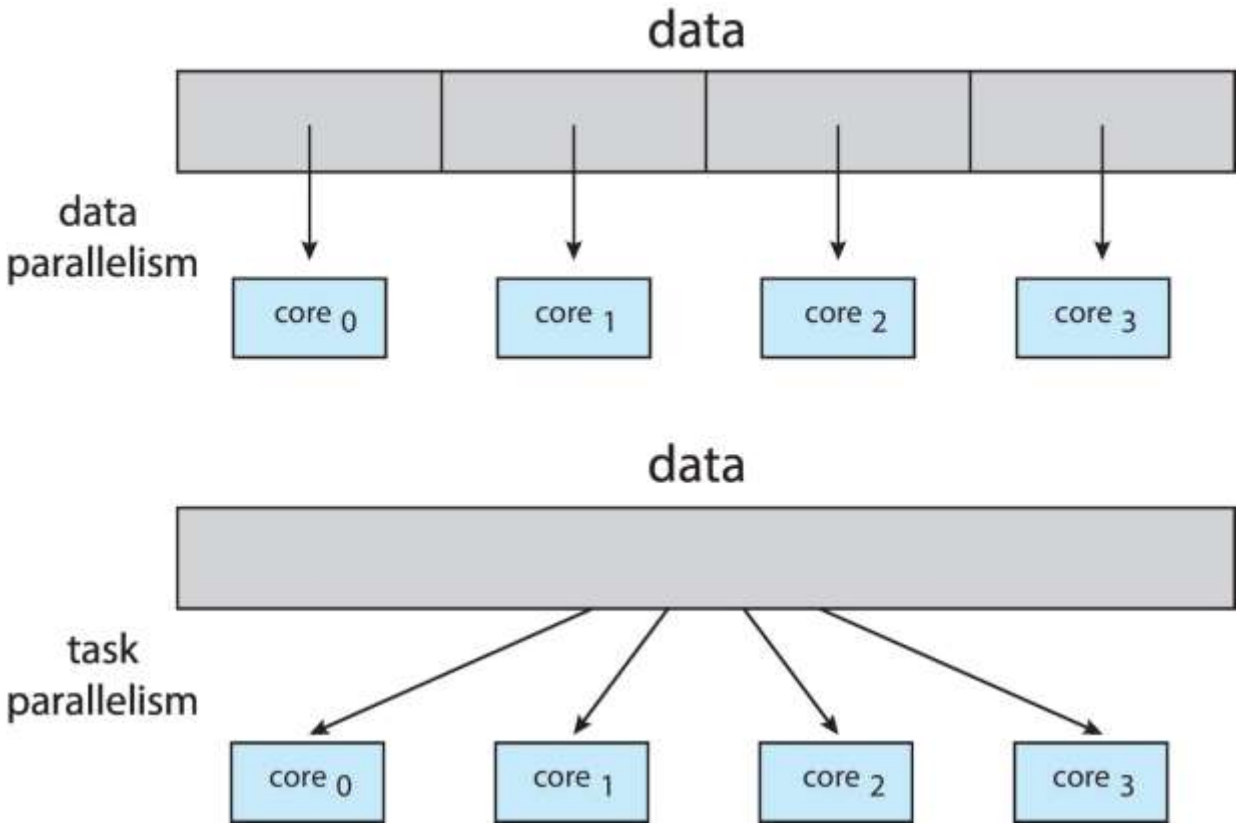
- **Multicore** or **multiprocessor** systems putting pressure on programmers, challenges include:
  - **Dividing activities**
  - **Balance**
  - **Data splitting**
  - **Data dependency**
  - **Testing and debugging**
- ***Parallelism*** implies a system can perform more than one task simultaneously
- ***Concurrency*** supports more than one task making progress
  - Single processor / core, scheduler providing concurrency

Multicore Programming (Cont.)

- Types of parallelism
  - **Data parallelism** – distributes subsets of the same data across multiple cores, same operation on each
  - **Task parallelism** – distributing threads across cores, each thread performing unique operation
- As # of threads grows, so does architectural support for threading
  - CPUs have cores as well as ***hardware threads***
  - Consider Oracle SPARC T4 with 8 cores, and 8 hardware threads per core

OPERATING SYSTEMS

Data and Task Parallelism



Amdahl's Law

- Identifies performance gains from adding additional cores to an application that has both serial and parallel components
- *S* is portion of an application that needs to be done in serial
- *N* processing cores

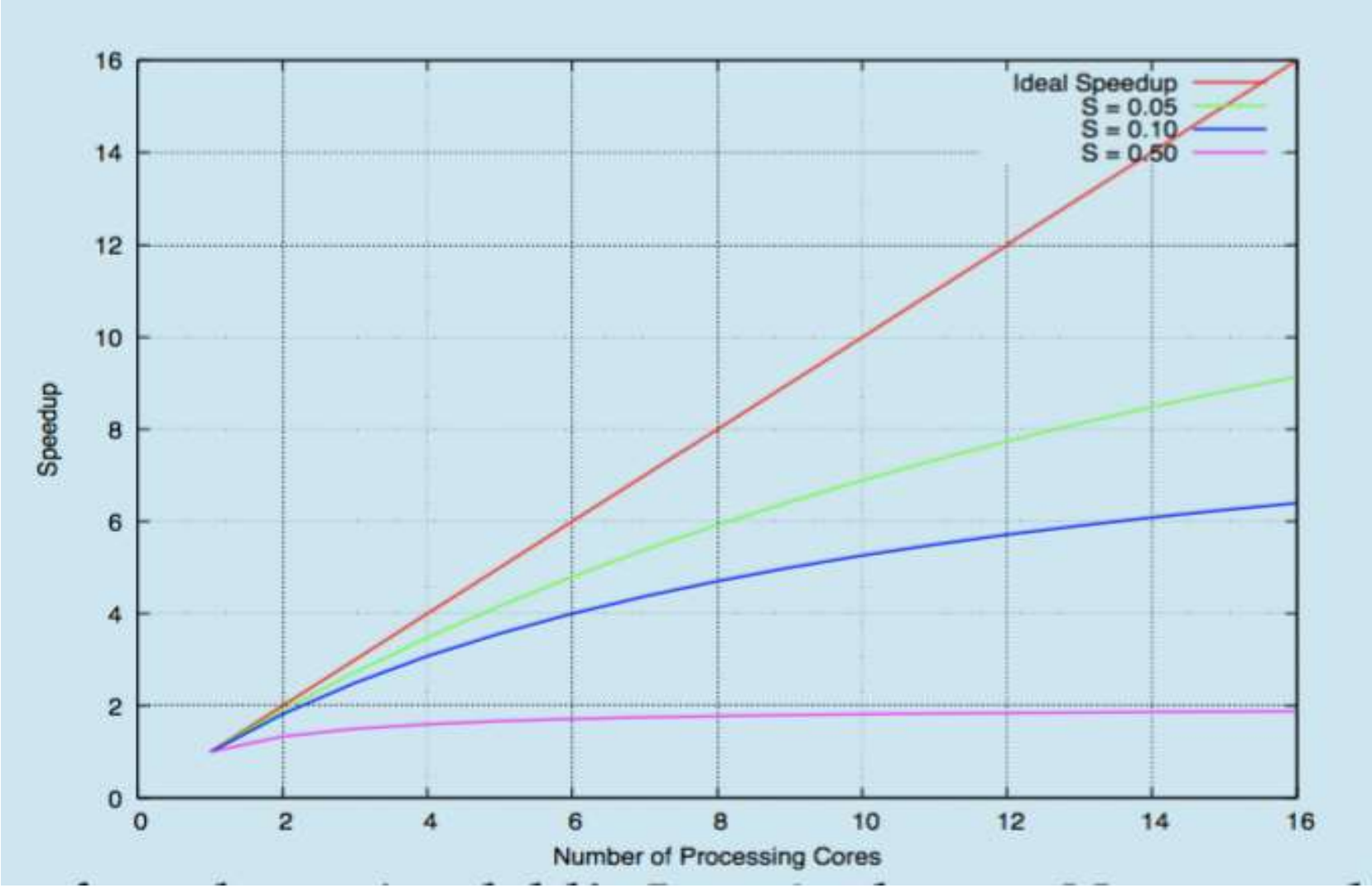
$$speedup \leq \frac{1}{S + \frac{(1-S)}{N}}$$

- That is, if application is 75% parallel / 25% serial, moving from 1 to 2 cores results in speedup of 1.6 times
- As *N* approaches infinity, speedup approaches 1 / *S*

  Serial portion of an application has disproportionate effect on performance gained by adding additional cores
- But does the law take into account contemporary multicore systems?

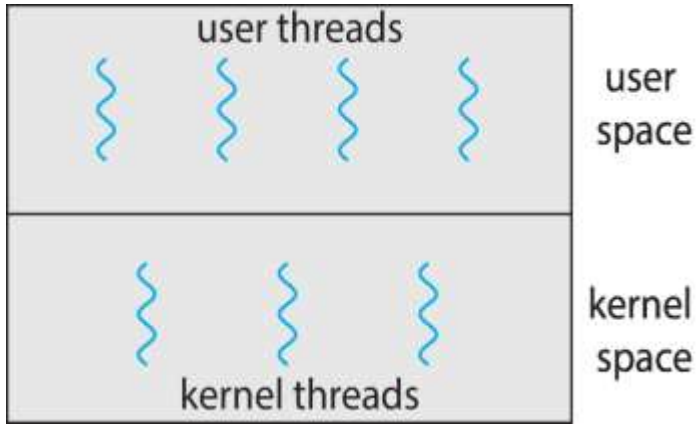
OPERATING SYSTEMS

Amdahl's Law



User Threads and Kernel Threads

- **User threads** - management done by user-level threads library
- Three primary thread libraries:
  - POSIX **Pthreads**
  - Windows threads
  - Java threads
- **Kernel threads** - Supported by the Kernel
- Examples – virtually all general purpose operating systems, including:
  - Windows
  - Solaris
  - Linux
  - Tru64 UNIX
  - Mac OS X



THANK YOU

Suresh Jamadagni
Department of Computer Science Engineering

sureshjamadagni@pes.edu

OPERATING SYSTEMS

Multithreading Models, Thread Creation and Scheduling

Suresh Jamadagni
Department of Computer Science

OPERATING SYSTEMS

Slides Credits for all the PPTs of this course

- The slides/diagrams in this course are an **adaptation**, **combination**, and **enhancement** of material from the following resources and persons:

1. Slides of Operating System Concepts, Abraham Silberschatz, Peter Baer Galvin, Greg Gagne - 9th edition 2013 and some slides from 10th edition 2018
2. Some conceptual text and diagram from Operating Systems - Internals and Design Principles, William Stallings, 9th edition 2018
3. Some presentation transcripts from A. Frank – P. Weisberg
4. Some conceptual text from Operating Systems: Three Easy Pieces, Remzi Arpaci-Dusseau, Andrea Arpaci Dusseau

OPERATING SYSTEMS

Multithreading Models, Thread Creation and Scheduling

Suresh Jamadagni
Department of Computer Science

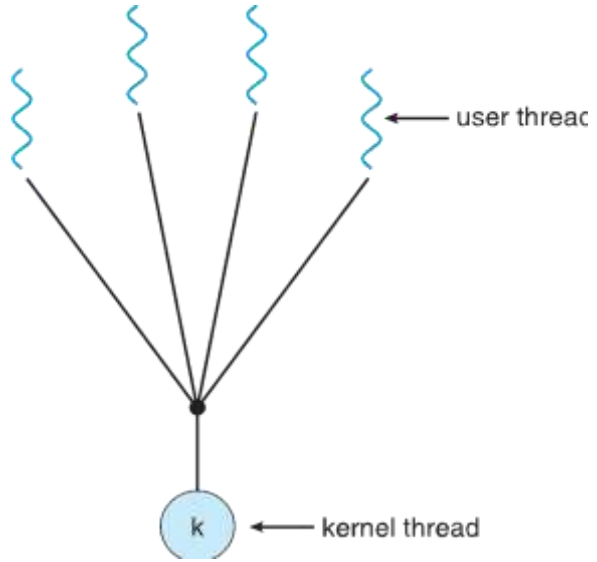
OPERATING SYSTEMS

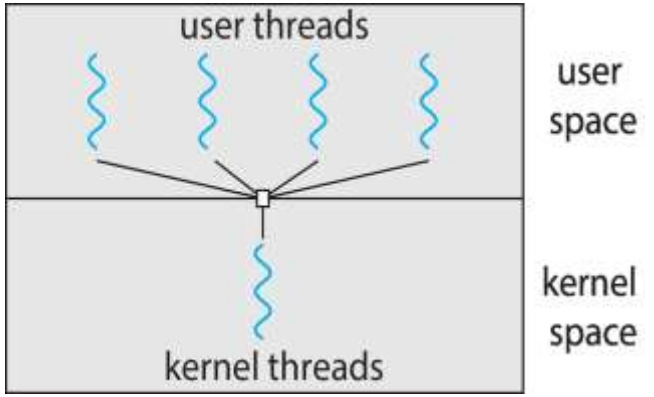
Multithreading Models

- Many-to-One
- One-to-One
- Many-to-Many

Many-to-One

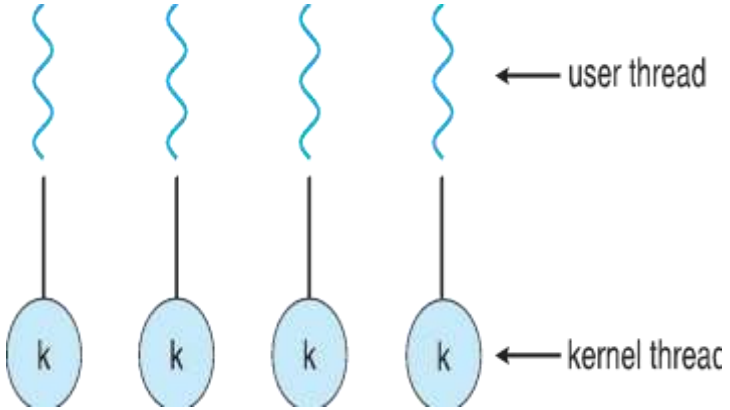
- Many user-level threads mapped to single kernel thread
- One thread blocking causes all to block
- Multiple threads may not run in parallel on multicore system because only one may be in kernel at a time
- Few systems currently use this model
- Examples:
  - **Solaris Green Threads**
  - **GNU Portable Threads**

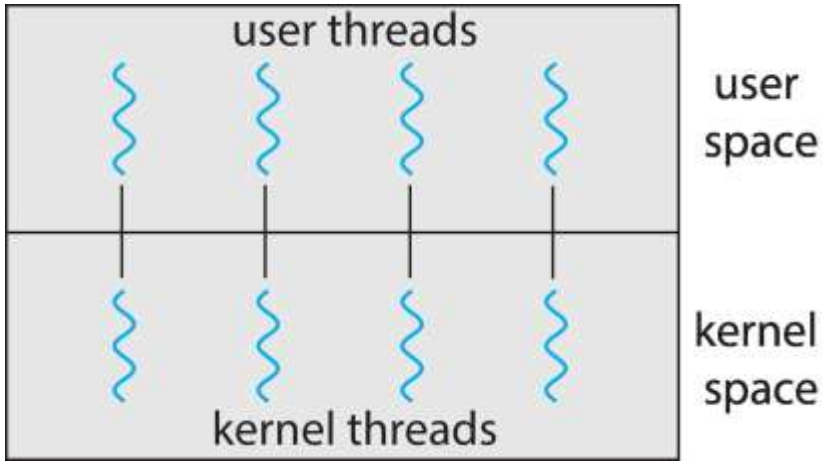




One-to-One

- Each user-level thread maps to kernel thread
- Creating a user-level thread creates a kernel thread
- More concurrency than many-to-one
- Number of threads per process sometimes restricted due to overhead
- Examples
  - Windows
  - Linux
  - Solaris 9 and later

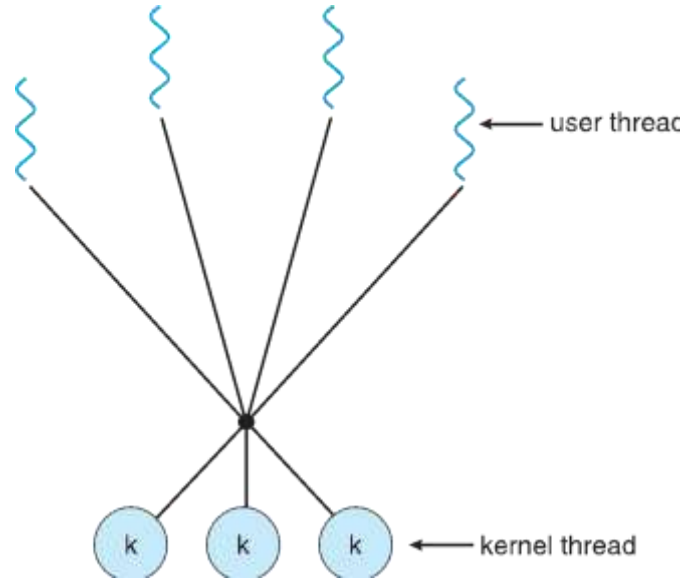


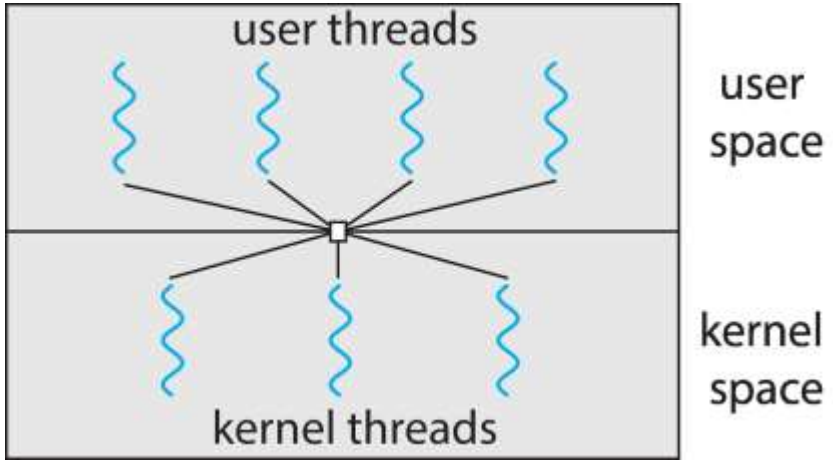


OPERATING SYSTEMS

Many-to-Many Model

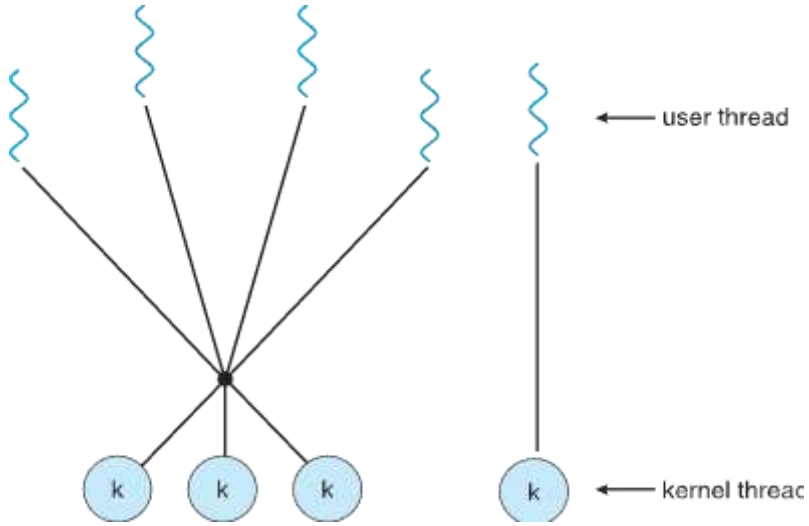
- Allows many user level threads to be mapped to many kernel threads
- Allows the operating system to create a sufficient number of kernel threads
- Solaris prior to version 9
- Windows with the *ThreadFiber* package

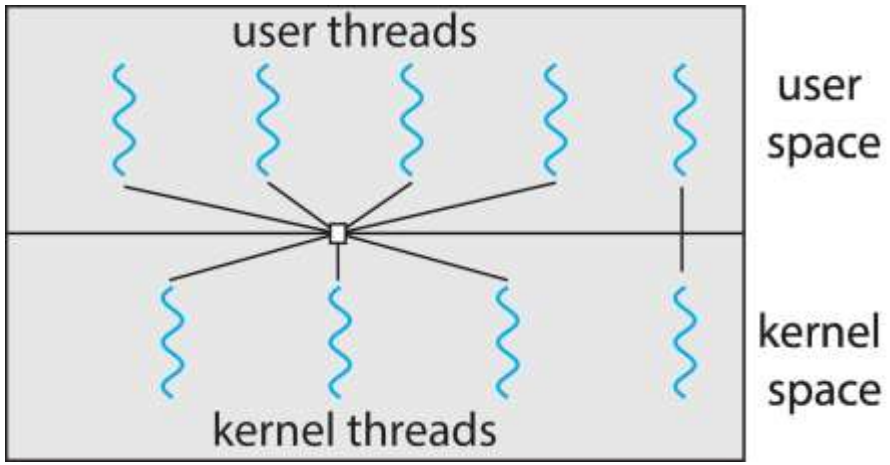




Two-level Model

- Similar to M:M, except that it allows a user thread to be **bound** to kernel thread
- Examples
  - IRIX
  - HP-UX
  - Tru64 UNIX
  - Solaris 8 and earlier





Pthreads

- May be provided either as user-level or kernel-level
- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
- ***Specification***, not ***implementation***
- API specifies behavior of the thread library, implementation is up to development of the library
- Common in UNIX operating systems (Solaris, Linux, Mac OS X)

OPERATING SYSTEMS

Pthreads Example

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

int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* threads call this function */

int main(int argc, char *argv[])
{
   pthread_t tid; /* the thread identifier */
   pthread_attr_t attr; /* set of thread attributes */

   if (argc != 2) {
      fprintf(stderr,"usage: a.out <integer value>\n");
      return -1;
   }
   if (atoi(argv[1]) < 0) {
      fprintf(stderr,"%d must be >= 0\n",atoi(argv[1]));
      return -1;
   }
```

Pthreads Example

```
   /* get the default attributes */
   pthread_attr_init(&attr);
   /* create the thread */
   pthread_create(&tid,&attr,runner,argv[1]);
   /* wait for the thread to exit */
   pthread_join(tid,NULL);

   printf("sum = %d\n",sum);
}

/* The thread will begin control in this function */
void *runner(void *param)
{
   int i, upper = atoi(param);
   sum = 0;

   for (i = 1; i <= upper; i++)
      sum += i;

   pthread_exit(0);
}
```

OPERATING SYSTEMS

Pthreads Code for Joining 10 Threads

```
#define NUM_THREADS 10

/* an array of threads to be joined upon */
pthread_t workers[NUM_THREADS];

for (int i = 0; i < NUM_THREADS; i++)
   pthread_join(workers[i], NULL);
```

Thread Scheduling

- Distinction between user-level and kernel-level threads
- When threads supported, threads scheduled, not processes
- Many-to-one and many-to-many models, thread library schedules user-level threads to run on LWP.
- Scheduling of user level threads (ULT) to kernel level threads (KLT) via leight-weight process (LWP) by the application developer.
  - **Leightweight Process (LWP) :**
    Light-weight process are threads in the user space that acts as an interface for the ULT to access the physical CPU resources
- Scheduling of kernel level threads by the system scheduler to perform different unique OS functions.

Thread Scheduling

- Many-to-one and many-to-many models, thread library schedules user-level threads to run on LWP
  - Known as **process-contention scope (PCS)** since scheduling competition is within the process
  - Typically done via priority set by programmer
- Kernel thread scheduled onto available CPU is **system-contention scope (SCS)** – competition among all threads in system

Pthread Scheduling

- API allows specifying either PCS or SCS during thread creation
  - PTHREAD_SCOPE_PROCESS schedules threads using PCS scheduling
  - PTHREAD_SCOPE_SYSTEM schedules threads using SCS scheduling
- Can be limited by OS – Linux and Mac OS X only allow PTHREAD_SCOPE_SYSTEM

OPERATING SYSTEMS

Pthread Scheduling API


```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5
int main(int argc, char *argv[]) {
   int i, scope;
   pthread_t tid[NUM THREADS];
   pthread_attr_t attr;
   /* get the default attributes */
   pthread_attr_init(&attr);
   /* first inquire on the current scope */
   if (pthread_attr_getscope(&attr, &scope) != 0)
      fprintf(stderr, "Unable to get scheduling scope\n");
   else {
      if (scope == PTHREAD_SCOPE_PROCESS)
         printf("PTHREAD_SCOPE_PROCESS");
      else if (scope == PTHREAD_SCOPE_SYSTEM)
         printf("PTHREAD_SCOPE_SYSTEM");
      else
         fprintf(stderr, "Illegal scope value.\n");
   }
```

Pthread Scheduling API

```
 /* set the scheduling algorithm to PCS or SCS */
   pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
   /* create the threads */
   for (i = 0; i < NUM_THREADS; i++)
      pthread_create(&tid[i],&attr,runner,NULL);
   /* now join on each thread */
   for (i = 0; i < NUM_THREADS; i++)
      pthread_join(tid[i], NULL);
}
/* Each thread will begin control in this function */
void *runner(void *param)
{
   /* do some work ... */
   pthread_exit(0);
}
```

THANK YOU

Suresh Jamadagni
Department of Computer Science Engineering

sureshjamadagni@pes.edu

OPERATING SYSTEMS

Threads and Concurrency

Suresh Jamadagni
Department of Computer Science

OPERATING SYSTEMS

Slides Credits for all the PPTs of this course

- The slides/diagrams in this course are an **adaptation**, **combination**, and **enhancement** of material from the following resources and persons:

1. Slides of Operating System Concepts, Abraham Silberschatz, Peter Baer Galvin, Greg Gagne - 9th edition 2013 and some slides from 10th edition 2018
2. Some conceptual text and diagram from Operating Systems - Internals and Design Principles, William Stallings, 9th edition 2018
3. Some presentation transcripts from A. Frank – P. Weisberg
4. Some conceptual text from Operating Systems: Three Easy Pieces, Remzi Arpaci-Dusseau, Andrea Arpaci Dusseau

OPERATING SYSTEMS

Pthreads and Windows Threads

Suresh Jamadagni
Department of Computer Science

Thread Libraries: Pthreads

- Provides the programmer with an API for creating and managing threads.
- Two primary ways of implementing a thread library.
  - Provide a library entirely in user space with no kernel support.
    - All code and data structures for the library exist in user space.
  - Implement a kernel-level library
    - Code and data structures for the library exist in kernel space.
    - Invoking a function directly by the operating system.

Thread Libraries: Pthreads

- Three main thread libraries are in use today:
  - (1) POSIX Pthreads,
  - (2)Win32,
  - (3) Java.
- Pthreads, the threads extension of the POSIX standard, may be provided as either a user- or kernel-level library.
- The Win32 thread library is a kernel-level library available on Windows systems.
- The Java thread API allows threads to be created and managed directly in Java programs.

Thread Libraries: Pthreads

- May be provided either as user-level or kernel-level
- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
- ***Specification***, not ***implementation***
- API specifies behavior of the thread library, implementation is up to development of the library
- Common in UNIX operating systems (Solaris, Linux, Mac OS X)

OPERATING SYSTEMS

Pthreads Example

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

int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* threads call this function */

int main(int argc, char *argv[])
{
   pthread_t tid; /* the thread identifier */
   pthread_attr_t attr; /* set of thread attributes */

   if (argc != 2) {
      fprintf(stderr,"usage: a.out <integer value>\n");
      return -1;
   }
   if (atoi(argv[1]) < 0) {
      fprintf(stderr,"%d must be >= 0\n",atoi(argv[1]));
      return -1;
   }
```

Pthreads Example

```
   /* get the default attributes */
   pthread_attr_init(&attr);
   /* create the thread */
   pthread_create(&tid,&attr,runner,argv[1]);
   /* wait for the thread to exit */
   pthread_join(tid,NULL);

   printf("sum = %d\n",sum);
}

/* The thread will begin control in this function */
void *runner(void *param)
{
   int i, upper = atoi(param);
   sum = 0;

   for (i = 1; i <= upper; i++)
      sum += i;

   pthread_exit(0);
}
```

OPERATING SYSTEMS

Pthreads Code for Joining 10 Threads

```
#define NUM_THREADS 10

/* an array of threads to be joined upon */
pthread_t workers[NUM_THREADS];

for (int i = 0; i < NUM_THREADS; i++)
   pthread_join(workers[i], NULL);
```

Windows threads

- Windows implements the Win32 API as its primary API.
- A Windows application runs as a separate process, and each process may contain one or more threads.
- Windows uses the one-to-one mapping.
- Windows also provides support for a **fiber** library, which provides the functionality of the many-to-many model

Windows threads

- The general components of a thread include:
  - A thread ID uniquely identifying the thread
  - A register set representing the status of the processor.
  - A user stack, employed when the thread is running in user mode, and a kernel stack, employed when the thread is running in kernel mode
  - A private storage area various run-time libraries and dynamic link libraries (DLLs)

Windows Threads

- Threads are created in theWin32 API using the CreateThread() function the Win32 API.
- Using the WaitForSingleObject() function,which causes the creating thread to block until the child thread has exited

OPERATING SYSTEMS

Windows Multithreaded C Program

```
#include <windows.h>
#include <stdio.h>
DWORD Sum; /* data is shared by the thread(s) */

/* the thread runs in this separate function */
DWORD WINAPI Summation(LPVOID Param)
{
   DWORD Upper = *(DWORD*)Param;
   for (DWORD i = 0; i <= Upper; i++)
      Sum += i;
   return 0;
}

int main(int argc, char *argv[])
{
   DWORD ThreadId;
   HANDLE ThreadHandle;
   int Param;

   if (argc != 2) {
      fprintf(stderr,"An integer parameter is required\n");
      return -1;
   }
   Param = atoi(argv[1]);
   if (Param < 0) {
      fprintf(stderr,"An integer >= 0 is required\n");
      return -1;
   }
```

Windows Multithreaded C Program (Cont.)

```
   /* create the thread */
   ThreadHandle = CreateThread(
      NULL, /* default security attributes */
      0, /* default stack size */
      Summation, /* thread function */
      &Param, /* parameter to thread function */
      0, /* default creation flags */
      &ThreadId); /* returns the thread identifier */

   if (ThreadHandle != NULL) {
       /* now wait for the thread to finish */
      WaitForSingleObject(ThreadHandle,INFINITE);

      /* close the thread handle */
      CloseHandle(ThreadHandle);

      printf("sum = %d\n",Sum);
   }
}
```

THANK YOU

Suresh Jamadagni
Department of Computer Science Engineering

sureshjamadagni@pes.edu

OPERATING SYSTEMS

Mutual Exclusion and Synchronization

Suresh Jamadagni
Department of Computer Science

OPERATING SYSTEMS

Software Approaches

Suresh Jamadagni
Department of Computer Science

OPERATING SYSTEMS

Slides Credits for all the PPTs of this course

- The slides/diagrams in this course are an **adaptation**, **combination**, and **enhancement** of material from the following resources and persons:

1. Slides of Operating System Concepts, Abraham Silberschatz, Peter Baer Galvin, Greg Gagne - 9th edition 2013 and some slides from 10th edition 2018
2. Some conceptual text and diagram from Operating Systems - Internals and Design Principles, William Stallings, 9th edition 2018
3. Some presentation transcripts from A. Frank – P. Weisberg
4. Some conceptual text from Operating Systems: Three Easy Pieces, Remzi Arpaci-Dusseau, Andrea Arpaci Dusseau

Background

- Processes can execute concurrently
  - May be interrupted at any time, partially completing execution
- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- Illustration of the problem:

  Suppose that we wanted to provide a solution to the consumer-producer problem that fills ***all*** the buffers. We can do so by having an integer **counter** that keeps track of the number of full buffers. Initially, **counter** is set to 0. It is incremented by the producer after it produces a new buffer and is decremented by the consumer after it consumes a buffer.

Producer

```
while (true) {
        /* produce an item in next_produced */

        while (counter == BUFFER_SIZE) ;
                /* do nothing */
        buffer[in] = next_produced;
        in = (in + 1) % BUFFER_SIZE; /* Variable in points to the next free position in the buffer */
        counter++;
}
```

Consumer

```
while (true) {
        while (counter == 0)
                ; /* do nothing */
        next_consumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE; /* Variable out points to the first full position in the buffer */
        counter--;
        /* consume the item in next consumed */
}
```

OPERATING SYSTEMS

Race Condition

- **counter++** could be implemented as

  register1 = counter
  register1 = register1 + 1
  counter = register1

- **counter--** could be implemented as

  register2 = counter
  register2 = register2 - 1
  counter = register2

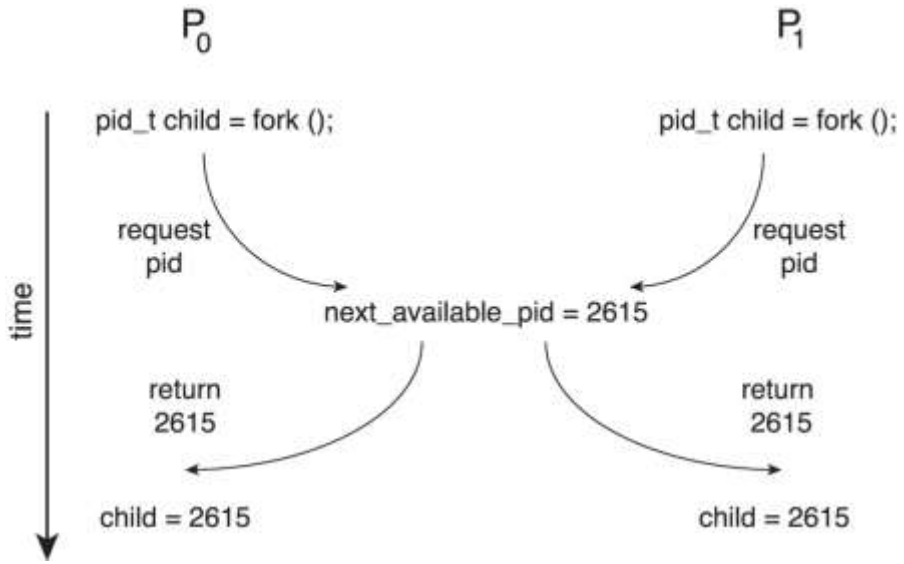
- Consider this execution interleaving with "count = 5" initially:

  S0: producer execute **register1 = counter** {register1 = 5}
  S1: producer execute **register1 = register1 + 1** {register1 = 6}
  S2: consumer execute **register2 = counter** {register2 = 5}
  S3: consumer execute **register2 = register2 – 1** {register2 = 4}
  S4: producer execute **counter = register1** {counter = 6 }
  S5: consumer execute **counter = register2** {counter = 4}

OPERATING SYSTEMS

Race Condition (Another Example)

- Processes P_0 and P_1 are creating child processes using the fork() system call
- Race condition on kernel variable next_available_pid which represents the next available process identifier (pid)



- Unless there is a mechanism to prevent P_0 and P_1 from accessing the variable next_available_pid the same pid could be assigned to two different processes!

Critical Section Problem

- Consider system of ***n*** processes {$p_0, p_1, \ldots p_{n-1}$}
- Each process has **critical section** segment of code
  - Process may be changing common variables, updating table, writing file, etc
  - When one process in critical section, no other may be in its critical section
- ***Critical section problem*** is to design protocol to solve this
- Each process must ask permission to enter critical section in **entry section**, may follow critical section with **exit section**, then **remainder section**

```
do {
        entry section
            critical section
        exit section
            remainder section
} while (true);
```

Critical Section

- General structure of process P_i

```
do {
        entry section
            critical section
        exit section
            remainder section
} while (true);
```

OPERATING SYSTEMS

Solution to Critical-Section Problem

1. **Mutual Exclusion** - If process P_i is executing in its critical section, then no other processes can be executing in their critical sections
2. **Progress** - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely
3. **Bounded Waiting** - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted
   - Assume that each process executes at a nonzero speed
   - No assumption concerning **relative speed** of the ***n*** processes

Critical-Section Handling in OS

Two approaches depending on if kernel is preemptive or non- preemptive

- **Preemptive** – allows preemption of process when running in kernel mode.
  - Preemptive kernels are difficult to design on SMP architectures
- **Non-preemptive** – runs until exits kernel mode, blocks, or voluntarily yields CPU
  - Essentially free of race conditions in kernel mode
  - It is free from race conditions on kernel data structures
- Preemptive kernel may be more responsive. Suitable for real-time systems.

OPERATING SYSTEMS

Peterson's Solution

- Software-based solution to the Critical Section problem.
- Good algorithmic description of solving the problem
- Assume that the **load** and **store** machine-language instructions are atomic; that is, cannot be interrupted
- **Peterson's Solution** restricted to two process.
- P_i and P_j are two process, j=1-i
- Peterson solution requires two processes share two data items:
  - int turn;
  - Boolean flag[2]
- The variable **turn** indicates whose turn it is to enter the critical section
- The **flag** array is used to indicate if a process is ready to enter the critical section. **flag[i] = *true*** implies that process $\mathbf{P_i}$ is ready!

Algorithm for Process P_i

```
while (true) {
        flag[i] = true;
        turn = j;
        while (flag[j] && turn = = j);
                /* critical section */
        flag[i] = false;
                /* remainder section */
}
```

The structure of process P_i in Peterson's solution

Software Solution

- To prove solution is correct, we need show
- Mutual exclusion is preserved

  $\mathbf{P_i}$ enters critical section only if:

  turn = i

  and **turn** cannot be both 0 and 1 at the same time
- What about the Progress requirement?
- What about the Bounded-waiting requirement?

Correctness of Peterson's Solution

- Provable that the three CS requirement are met:
  1. Mutual exclusion is preserved

     $\mathbf{P_i}$ enters CS only if:

     either **flag[j] = false** or **turn = i**
  2. Progress requirement is satisfied
  3. Bounded-waiting requirement is met

THANK YOU

Suresh Jamadagni
Department of Computer Science Engineering

sureshjamadagni@pes.edu

OPERATING SYSTEMS

Synchronization

Suresh Jamadagni
Department of Computer Science

OPERATING SYSTEMS

Slides Credits for all the PPTs of this course

- The slides/diagrams in this course are an **adaptation**, **combination**, and **enhancement** of material from the following resources and persons:

1. Slides of Operating System Concepts, Abraham Silberschatz, Peter Baer Galvin, Greg Gagne - 9th edition 2013 and some slides from 10th edition 2018
2. Some conceptual text and diagram from Operating Systems - Internals and Design Principles, William Stallings, 9th edition 2018
3. Some presentation transcripts from A. Frank – P. Weisberg
4. Some conceptual text from Operating Systems: Three Easy Pieces, Remzi Arpaci-Dusseau, Andrea Arpaci Dusseau

OPERATING SYSTEMS

Principles of concurrency
Synchronization Hardware

Suresh Jamadagni
Department of Computer Science

Review of Peterson's solution

Code for process i

```
do{
flag[i]=TRUE
turn=j
while(flag[j]&&turn==j);//Do-nop
   critical section
flag[i]=FALSE;
Reminder section
}while(TRUE)
```

Code for process j

```
do{
flag[j]=TRUE
turn=i
while(flag[i]&&turn==i);//Do-nop
   critical section
flag[j]=FALSE;
Reminder section
}while(TRUE)
```

Principles of Concurrency

- relative speed of execution of processes is not predictable.
- system interrupts are not predictable
- scheduling policies may vary

Synchronization Hardware

- Software based solutions are not guaranteed to work on modern computer architectures
- Many systems provide hardware support for implementing the critical section code.
- All solutions below based on idea of **locking**
  - Protecting critical regions via locks.
- synchronization can be done through Lock & Unlock technique
- Locking part is done in the Entry Section. After locking the process enter critical section.
- The process is moved to the Exit Section after it is done with execution in CS.
- Unlock is done in exit section.
- This process is designed in such a way that all the three conditions of the Critical Sections are satisfied

Synchronization Hardware

- Uniprocessors – could disable interrupts
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems
    - Operating systems using this not broadly scalable
- Modern machines provide special atomic hardware instructions
    - **Atomic** = non-interruptible
  - Either test memory word and set value
  - Or swap contents of two memory words

test_and_set Instruction

- Test and Set Lock (TSL) is a synchronization mechanism.
- It uses a test and set instruction to provide the synchronization among the processes executing concurrently.
- It is an instruction that returns the old value of a memory location and sets the memory location value to 1 as a single atomic operation.
- If one process is currently executing a test-and-set, no other process is allowed to begin another test-and-set until the first process test-and-set is finished.

test_and_set Instruction

Definition:

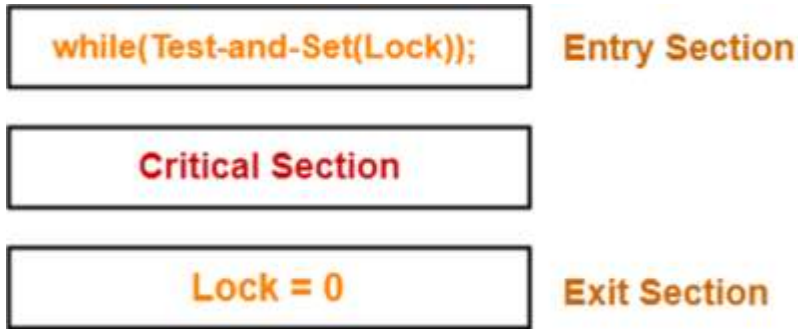
```
boolean test_and_set (boolean *target)
  {
     boolean rv = *target;
     *target = TRUE;
     return rv:
  }
```

1. Executed atomically
2. Returns the original value of passed parameter
3. Set the new value of passed parameter to “TRUE”.

Solution using test_and_set()

- Shared Boolean variable lock, initialized to FALSE
- Solution:

```
do {
   while (test_and_set(&lock))
      ; /* do nothing */
         /* critical section */
   lock = false;
         /* remainder section */
} while (true);
```



OPERATING SYSTEMS

compare_and_swap Instruction

Definition:

```
int compare _and_swap(int *value, int expected, int new_value) {
   int temp = *value;

   if (*value == expected)
      *value = new_value;
return temp;
}
```

```
do{
while(compare_and_swap(&lock,0,1)!=0);
Critical section
lock=0
Remainder section
}while(true)
```

1. Executed atomically
2. Returns the original value of passed parameter “value”
3. Set the variable “value” the value of the passed parameter “new_value” but only if *value == expected. That is, the swap takes place only under this condition.
4. In the x86 (since 80486) and Itanium architectures this is implemented as compare and exchange (CMPXCHG) instruction

Solution using compare_and_swap()

- Shared integer "lock" initialized to 0;
- Solution:

```
do {
      while (compare_and_swap(&lock, 0, 1) != 0)
    ; /* do nothing */
   /* critical section */
 lock = 0;
   /* remainder section */
} while (true);
```

Mutual exclusion is satisfied

Do not satisfy bounded waiting requirement

Bounded-waiting Mutual Exclusion with test_and_set

This test_and_set algorithm satisfies all the critical section requirements

The common data structures are

boolean waiting[n];

boolean lock;

```
do {
    waiting[i] = true;
    key = true;
    while (waiting[i] && key)
        key = test_and_set(&lock);
    waiting[i] = false;

/* critical section */

    j = (i + 1) % n;
    while ((j != i) && !waiting[j])
        j = (j + 1) % n;
    if (j == i)
        lock = false;
    else
        waiting[j] = false;

/* remainder section */

} while (true);
```

THANK YOU

Suresh Jamadagni
Department of Computer Science and Engineering

sureshjamadagni@pes.edu

OPERATING SYSTEMS

Mutex Locks and Semaphores

Suresh Jamadagni
Department of Computer Science

OPERATING SYSTEMS

Slides Credits for all the PPTs of this course

- The slides/diagrams in this course are an **adaptation**, **combination**, and **enhancement** of material from the following resources and persons:

1. Slides of Operating System Concepts, Abraham Silberschatz, Peter Baer Galvin, Greg Gagne - 9th edition 2013 and some slides from 10th edition 2018
2. Some conceptual text and diagram from Operating Systems - Internals and Design Principles, William Stallings, 9th edition 2018
3. Some presentation transcripts from A. Frank – P. Weisberg
4. Some conceptual text from Operating Systems: Three Easy Pieces, Remzi Arpaci-Dusseau, Andrea Arpaci Dusseau

Mutex Locks

- Previous solutions i.e hardware and software solutions are complicated and generally inaccessible to application programmers
- OS designers build software tools to solve critical section problem
- Simplest is mutex lock
- Protect a critical section by first **acquire()** a lock then **release()** the lock
  - Boolean variable indicating if lock is available or not
- Calls to **acquire()** and **release()** must be atomic
  - Usually implemented via hardware atomic instructions
- But this solution requires **busy waiting**
- This lock therefore called a **spinlock**
- It is problem in real time systems
- Busy waiting wastes CPU cycles

Advantages of spinlocks

- no context switch is required when a process must wait on a lock.
- context switch may take considerable time.
- When locks are expected to be held for short times, spinlocks are useful
- These are employed on multiprocessor systems where one thread can "spin" on one processor while another thread performs its critical section on another processor.

Solution to Critical-section Problem Using Locks

```
while (TRUE) {
    acquire lock
        critical section
    release lock
remainder section
}
```

- **Implementation of acquire and release**

```
acquire() {
    while (!available)
        ; /* busy wait */
    available = false;
}
release() {
    available = true;
}
```

acquire() and release()

- Solution to a critical section problem using mutex
- **do {**

  acquire lock

  critical section

  release lock

  remainder section

} while (true);

OPERATING SYSTEMS

Scenario 2

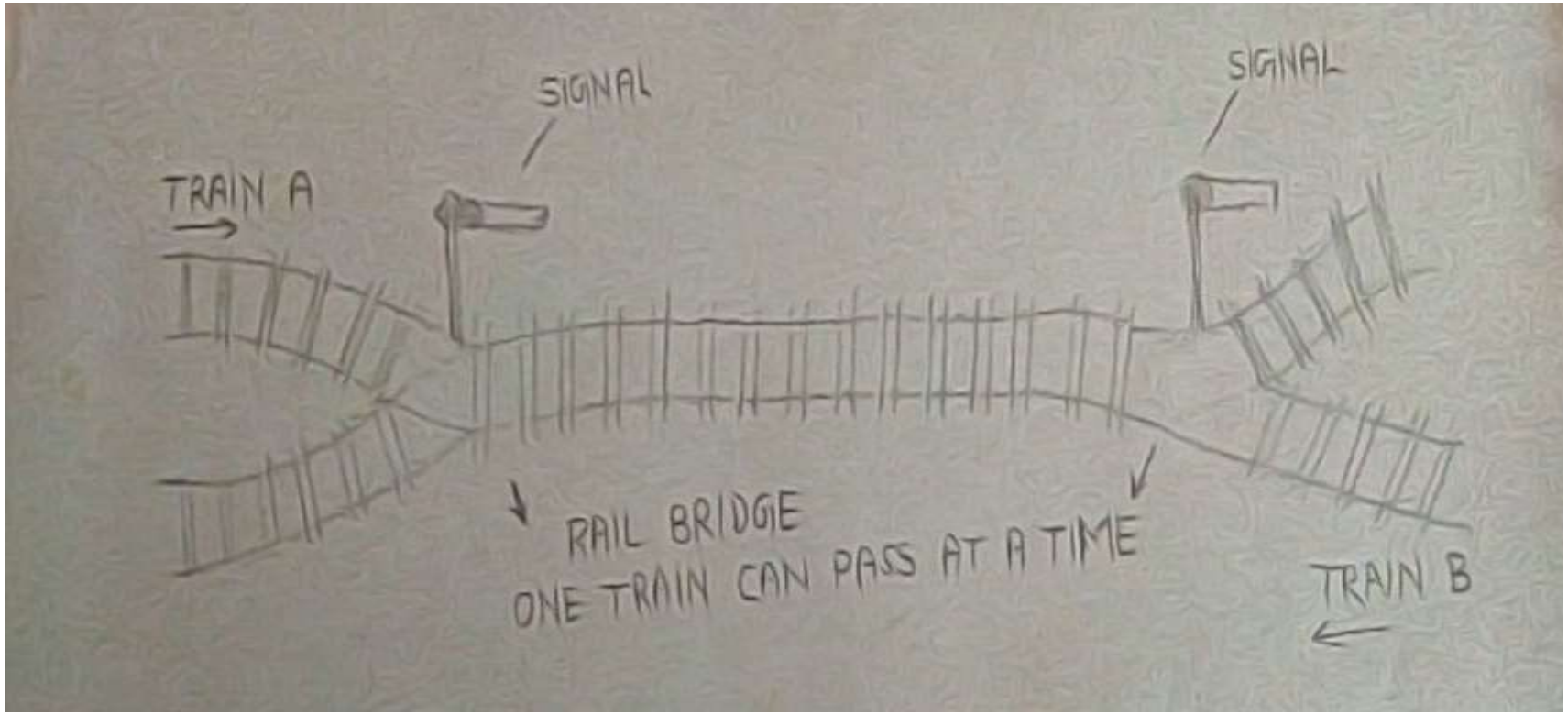


Image src: https://www.linkedin.com/pulse/concurrency-semaphore-real-life-kunal-saxena/

Scenario 1

- Consider a library of an university with 10 rooms
- At a time one room can be used by only one student by informing the front desk for reading.
- Once he completes reading, he has to inform the front desk.
- Person at front desk knows how many rooms are available for use and how many are occupied, how may of them are waiting.
- Once the room is vacant, who will get the chance to occupy the room?

Idea Src:https://en.wikipedia.org/wiki/Semaphore_(programming)

Semaphore

- Synchronization tool that provides more sophisticated ways (than Mutex locks) for process to synchronize their activities.
- Semaphore ***S*** – integer variable
- Can only be accessed via two indivisible (atomic) operations
  - **wait()** and **signal()**
    - Originally called **P()** and **V()**
- Definition of the **wait() operation**

```
wait(S) {
   while (S <= 0)
     ; // busy wait
   S--;}
```

- Definition of the **signal() operation**

```
signal(S) {
   S++;
}
```

Semaphore

- **Counting semaphore** – integer value can range over an unrestricted domain
- **Binary semaphore** – integer value can range only between 0 and 1
  - Same as a **mutex lock**
- Can solve various synchronization problems
- Consider P_1 and P_2 that require S_1 to happen before S_2

  Create a semaphore "**synch**" initialized to 0

```
P1:
  S1;
  signal(synch);
P2:
  wait(synch);
  S2;
```

- Can implement a counting semaphore ***S*** as a binary semaphore

Semaphore Implementation

- Must guarantee that no two processes can execute the **wait()** and **signal()** on the same semaphore at the same time
- Thus, the implementation becomes the critical section problem where the **wait** and **signal** code are placed in the critical section
  - Could now have **busy waiting** in critical section implementation
    - But implementation code is short
    - Little busy waiting if critical section rarely occupied
- Note that applications may spend lots of time in critical sections and therefore this is not a good solution

Semaphore Implementation with no Busy waiting

- With each semaphore there is an associated waiting queue
- Each entry in a waiting queue has two data items:
  - value (of type integer)
  - pointer to next record in the list
- Two operations:
  - **block** – place the process invoking the operation on the appropriate waiting queue
  - **wakeup** – remove one of processes in the waiting queue and place it in the ready queue
- **typedef struct{**

int value;

struct process *list;

} semaphore;

Semaphore Implementation with no Busy waiting (Cont.)

```
wait(semaphore *S) {
  S->value--;
  if (S->value < 0) {
    add this process to S->list;
    block();
  }
}
signal(semaphore *S) {
  S->value++;
  if (S->value >= 0) {
    remove a process P from S->list;
    wakeup(P);
  } }
```

OPERATING SYSTEMS

Problems with Semaphores

- Incorrect use of semaphore operations:
  - signal (mutex) …. wait (mutex)
  - wait (mutex) … wait (mutex)
  - Omitting of wait (mutex) or signal (mutex) (or both)
- Deadlock and starvation are possible.
- **Deadlock** – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
- Let ***S*** and ***Q*** be two semaphores initialized to 1

P_0

```
wait(S);
wait(Q);
 ...
signal(S);
signal(Q);
```

P_1

```
wait(Q);
wait(S);
 ...
signal(Q);
signal(S);
```

Problems with Semaphores (Cont.)

- **Starvation** – **indefinite blocking**
  - A process may never be removed from the semaphore queue in which it is suspended
- **Priority Inversion** – Scheduling problem when lower-priority process holds a lock needed by higher-priority process
  - Solved via **priority-inheritance protocol**

Priority Inheritance Protocol

- When several tasks are waiting for the same critical resource, the task which is currently holding this critical resource is given the highest priority among all the tasks which are waiting for the same critical resource.
- Now after the lower priority task having the critical resource is given the highest priority then the intermediate priority tasks can not preempt this task. This helps in avoiding priority inversion.
- When the task which is given the highest priority among all tasks, finishes the job and releases the critical resource then it gets back to its original priority value (which may be less or equal).
- It allows the different priority tasks to share the critical resources.

Priority Inheritance Protocol

- Consider the scenario with three processes **P1, P2**, and **P3**.
  - **P1** has the highest priority, **P2** the next highest, and **P3** the lowest.
- Assume that **P3** is holding semaphore **S** and that **P1** is waiting for **S** to be released
- Assume that **P2** is assigned the CPU and preempts **P3**
  - **P3** is still holding semaphore **S**
  - **P1** is waiting for **S** to be released
- What has happened is that **P2** - a process with a lower priority than **P1** - has indirectly prevented **P3** from gaining access to the resource.
- To prevent this from occurring, a **priority inheritance protocol** is used.

THANK YOU

Suresh Jamadagni
Department of Computer Science and Engineering

sureshjamadagni@pes.edu

OPERATING SYSTEMS

Classic problems of Synchronization

Chandravva Hebbi
Department of Computer Science

OPERATING SYSTEMS

Slides Credits for all the PPTs of this course

- The slides/diagrams in this course are an **adaptation**, **combination**, and **enhancement** of material from the following resources and persons:

1. Slides of Operating System Concepts, Abraham Silberschatz, Peter Baer Galvin, Greg Gagne - 9th edition 2013 and some slides from 10th edition 2018
2. Some conceptual text and diagram from Operating Systems - Internals and Design Principles, William Stallings, 9th edition 2018
3. Some presentation transcripts from A. Frank – P. Weisberg
4. Some conceptual text from Operating Systems: Three Easy Pieces, Remzi Arpaci-Dusseau, Andrea Arpaci Dusseau

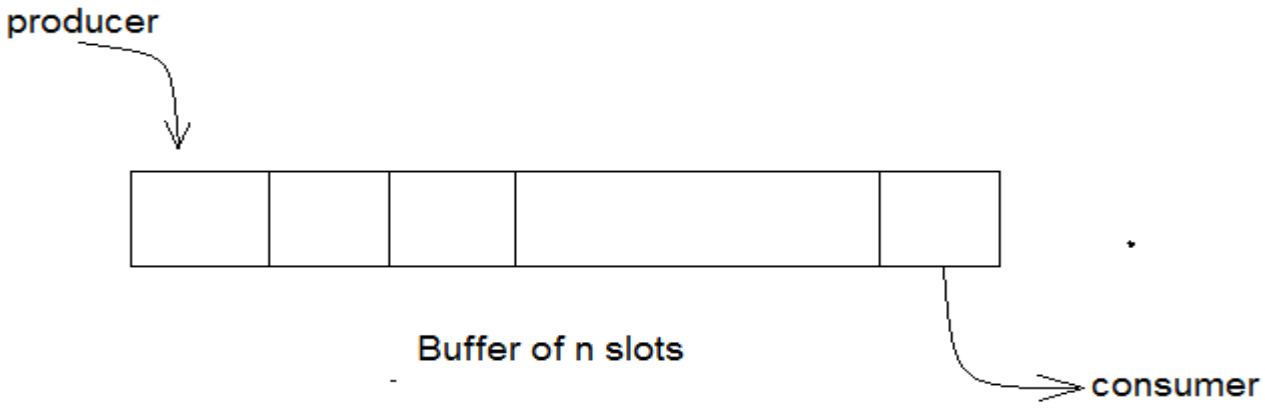
OPERATING SYSTEMS

- **Classical problems of Synchronization**

Suresh Jamadagni
Department of Computer Science

Bounded-Buffer Problem

- ***n*** buffers, each can hold one item
- Semaphore **mutex** initialized to the value 1
- Semaphore **full** initialized to the value 0
- Semaphore **empty** initialized to the value n



Bounded-Buffer Problem (Cont.)

- The structure of the producer process

```
do {
    ...
    /* produce an item in next_produced */
    ...
    wait(empty); //wait until empty > 0 and then decrement 'empty'
    wait(mutex); //acquire lock
    ...
    /* add next produced to the buffer */
    ...
    signal(mutex); //release a lock
    signal(full); //increment full
} while (true);
```

Bounded-Buffer Problem (Cont.)

- The structure of the consumer process

```
do {
  wait(full); // wait until full > 0 and then decrement 'full'
  wait(mutex); // acquire the lock
  ...
  /* remove an item from buffer to next_consumed */
  ...
  signal(mutex); // release the lock
  signal(empty); // increment 'empty'
  ...
  /* consume the item in next consumed */
  ...
} while (true);
```

Readers-Writers Problem

The Problem Statement

There is a shared resource which should be accessed by multiple processes. There are two types of processes in this context. They are **reader** and **writer**. Any number of **readers** can read from the shared resource simultaneously, but only one **writer** can write to the shared resource. When a **writer** is writing data to the resource, no other process can access the resource. A **writer** cannot write to the resource if there are non zero number of readers accessing the resource at that time.

Readers-Writers Problem

- **solution**
- Several variations of how readers and writers are considered – all involve some form of priorities
- Shared Data
  - Data set
  - Semaphore **rw_mutex** initialized to 1(semaphore)
  - Semaphore **mutex** initialized to 1 (mutex)
  - Integer **read_count** initialized to 0

Readers-Writers Problem (Cont.)

- The structure of a writer process

```
do {
        wait(rw_mutex);
        ...
        /* writing is performed */
        ...
        signal(rw_mutex);
} while (true);
```

Readers-Writers Problem (Cont.)

- The structure of a reader process

```
do {
        wait(mutex);
        read_count++;
        if (read_count == 1)
            wait(rw_mutex);
        signal(mutex);
        ...
        /* reading is performed */
        ...
        wait(mutex);
        read count--;
        if (read_count == 0)
            signal(rw_mutex);
        signal(mutex);
} while (true);
```

Readers-Writers Problem Variations

- ***First*** variation – no reader kept waiting unless writer has permission to use shared object
- ***Second*** variation – once writer is ready, it performs the write ASAP
- Both may have starvation leading to even more variations
- Problem is solved on some systems by kernel providing reader-writer locks

Dining-Philosophers Problem

- Philosophers spend their lives alternating thinking and eating
- Don’t interact with their neighbors, occasionally try to pick up 2 chopsticks (one at a time) to eat from bowl
  - Need both to eat, then release both when done
- In the case of 5 philosophers
  - Shared data
    - Bowl of rice (data set)
    - Semaphore chopstick [5] initialized to 1

Dining-Philosophers Problem Algorithm

- The structure of Philosopher *i*:

```
do {
    wait (chopstick[i] );
    wait (chopStick[ (i + 1) % 5] );
            //  eat
    signal (chopstick[i] );
    signal (chopstick[ (i + 1) % 5] );
            //  think
} while (TRUE);
```

- What is the problem with this algorithm?

Dining-Philosophers Problem Algorithm (Cont.)

- Deadlock handling
  - Allow at most 4 philosophers to be sitting simultaneously at the table.
  - Allow a philosopher to pick up the chopsticks only if both are available (picking must be done in a critical section).
  - Use an asymmetric solution -- an odd-numbered philosopher picks up first the left chopstick and then the right chopstick. Even-numbered philosopher picks up first the right chopstick and then the left chopstick.

THANK YOU

Suresh Jamadagni
Department of Computer Science Engineering

sureshjamadagni@pes.edu

OPERATING SYSTEMS

Synchronization Examples

Suresh Jamadagni
Department of Computer Science

OPERATING SYSTEMS

Slides Credits for all the PPTs of this course

- The slides/diagrams in this course are an **adaptation**, **combination**, and **enhancement** of material from the following resources and persons:

1. Slides of Operating System Concepts, Abraham Silberschatz, Peter Baer Galvin, Greg Gagne - 9th edition 2013 and some slides from 10th edition 2018
2. Some conceptual text and diagram from Operating Systems - Internals and Design Principles, William Stallings, 9th edition 2018
3. Some presentation transcripts from A. Frank – P. Weisberg
4. Some conceptual text from Operating Systems: Three Easy Pieces, Remzi Arpaci-Dusseau, Andrea Arpaci Dusseau

OPERATING SYSTEMS

Synchronization Examples

- Windows
- Linux
- Pthreads
- Solaris

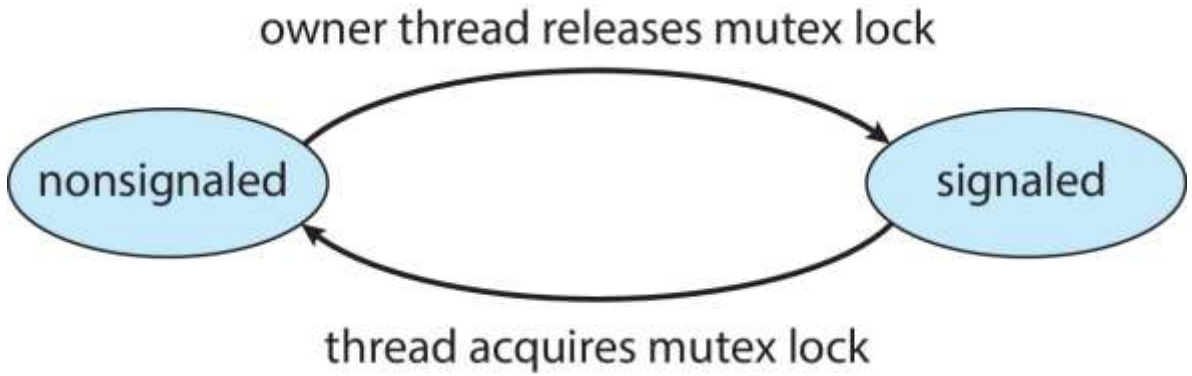
Suresh Jamadagni
Department of Computer Science

Windows Synchronization

- Uses interrupt masks to protect access to global resources on uniprocessor systems
- Uses **spinlocks** on multiprocessor systems
  - For reasons of efficiency, kernel ensures that a thread will never be preempted while holding a spinlock.
- Also provides **dispatcher objects** outside the kernel, to synchronize mutex locks, semaphores, events, and timers
  - **Events**
    - An event acts much like a condition variable (i.e notify a waiting thread when a desired condition occurs)
  - Timers notify one or more thread when time expired

Windows Synchronization - Mutex dispatcher object

- Dispatcher objects may be in either a **signaled-state (**object available and a thread will not block) or a **non-signaled state** (object not available, thread will block)
- A Relationship exists between the state of a dispatcher object and the state of a thread.
  - State of a thread changes from ready to waiting and vice-versa



Linux Synchronization

- Linux:
  - Prior to kernel Version 2.6, disables interrupts to implement short critical sections
  - Version 2.6 and later, fully preemptive
- Linux provides:
  - Semaphores
  - atomic integers
  - spinlocks
  - reader-writer versions of both
- On single-cpu system, spinlocks replaced by enabling and disabling kernel preemption

Linux Synchronization (Cont.)

- Atomic variables
  atomic_t is the data type for atomic integer
- Consider the variables
  atomic_t counter;
  int value;

Atomic Operation	*Effect*
`atomic_set(&counter,5);`	`counter = 5`
`atomic_add(10,&counter);`	`counter = counter + 10`
`atomic_sub(4,&counter);`	`counter = counter - 4`
`atomic_inc(&counter);`	`counter = counter + 1`
`value = atomic_read(&counter);`	`value = 12`

Pthreads (POSIX) Synchronization

- Pthreads API is OS-independent, widely used on UNIX, Linux, and macOS
- It provides:
  - mutex locks
  - semaphores
  - condition variable
- Non-portable extensions include:
  - read-write locks
  - spinlocks

POSIX Mutex Locks

- Creating and initializing the lock

```
#include <pthread.h>

pthread_mutex_t mutex;

/* create and initialize the mutex lock */
pthread_mutex_init(&mutex,NULL);
```

- Acquiring and releasing the lock

```
/* acquire the mutex lock */
pthread_mutex_lock(&mutex);

/* critical section */

/* release the mutex lock */
pthread_mutex_unlock(&mutex);
```

POSIX Semaphores

- POSIX provides two versions – **named** and **unnamed**.
- Named semaphores (have actual names in the file system) can be shared by multiple unrelated processes
- Unnamed semaphores can be used only by threads belonging to the same process.

POSIX Named Semaphores

- Creating and initializing the semaphore:

```
#include <semaphore.h>
sem_t *sem;

/* Create the semaphore and initialize it to 1 */
sem = sem_open("SEM", O_CREAT, 0666, 1);
```

- Another process can access the semaphore by referring to its name **SEM**.
- Acquiring and releasing the semaphore:

```
/* acquire the semaphore */
sem_wait(sem);

/* critical section */

/* release the semaphore */
sem_post(sem);
```

OPERATING SYSTEMS

POSIX Unnamed Semaphores

- Creating and initializing the semaphore:

```
#include <semaphore.h>
sem_t sem;

/* Create the semaphore and initialize it to 1 */
sem_init(&sem, 0, 1);
```

- Acquiring and releasing the semaphore:

```
/* acquire the semaphore */
sem_wait(&sem);

/* critical section */

/* release the semaphore */
sem_post(&sem);
```

POSIX Condition Variables

- Since POSIX is typically used in C/C++ and these languages do not provide a monitor (A high-level abstraction that provides a convenient and effective mechanism for process synchronization) , POSIX condition variables are associated with a POSIX mutex lock to provide mutual exclusion: Creating and initializing the condition variable:

```
pthread_mutex_t mutex;
pthread_cond_t cond_var;

pthread_mutex_init(&mutex,NULL);
pthread_cond_init(&cond_var,NULL);
```

OPERATING SYSTEMS

POSIX Condition Variables

- Thread waiting for the condition **a == b** to become true:

```
pthread_mutex_lock(&mutex);
while (a != b)
     pthread_cond_wait(&cond_var, &mutex);

pthread_mutex_unlock(&mutex);
```

- Thread signaling another thread waiting on the condition variable:

```
pthread_mutex_lock(&mutex);
a = b;
pthread_cond_signal(&cond_var);
pthread_mutex_unlock(&mutex);
```

Solaris Synchronization

- Implements a variety of locks to support multitasking, multithreading (including real-time threads), and multiprocessing
- Uses **adaptive mutexes** for efficiency when protecting data from short code segments (< a few 100 instructions)
  - Starts as a standard semaphore spin-lock
  - If lock held, and by a thread running on another CPU, spins
  - If lock held by non-run-state thread (i.e. the thread holding the lock is not currently in run state), block and sleep waiting for signal of lock being released

Solaris Synchronization (Cont.)

- Uses **condition variables**
- Uses **readers-writers** locks when longer sections of code need access to data
- Uses **turnstiles** to order the list of threads waiting to acquire either an adaptive mutex or reader-writer lock
  - Turnstiles are per-lock-holding-thread, not per-object
- Priority-inheritance per-turnstile gives the running thread the highest of the priorities of the threads in its turnstile

THANK YOU

Suresh Jamadagni
Department of Computer Science and Engineering

sureshjamadagni@pes.edu

OPERATING SYSTEMS

Deadlocks

Suresh Jamadagni
Department of Computer Science

OPERATING SYSTEMS

Slides Credits for all the PPTs of this course

- The slides/diagrams in this course are an **adaptation**, **combination**, and **enhancement** of material from the following resources and persons:

1. Slides of Operating System Concepts, Abraham Silberschatz, Peter Baer Galvin, Greg Gagne - 9th edition 2013 and some slides from 10th edition 2018
2. Some conceptual text and diagram from Operating Systems - Internals and Design Principles, William Stallings, 9th edition 2018
3. Some presentation transcripts from A. Frank – P. Weisberg
4. Some conceptual text from Operating Systems: Three Easy Pieces, Remzi Arpaci-Dusseau, Andrea Arpaci Dusseau

OPERATING SYSTEMS

Principles of Deadlocks, Deadlock Characterization

Suresh Jamadagni
Department of Computer Science

Introduction

- Multiprogramming environment: several processes compete to limited number of resources
- A Process is holding a resource(R1) and is waiting for the resources(R2).
- The resource R2 is held by another process..
- Waiting state of processes will not change, as the requested resource is held by the waiting process.
- This situation is called deadlock

System Model

- System consists of finite number of resources
- Resource types R_1, R_2, . . ., R_m
  - *Physical resources: CPU cycles, memory space, I/O devices, printer, tape drives.*
  - *Logical resources: semaphores, mutex locks, files.*
- Each resource type R_i has W_i instances.
- Ex: if the system has 2 CPU's then CPU has 2 instances
- Each process utilizes a resource as follows:
  - **Request :** Process makes request to the resource. Eg. system call like request(), open(), wait(),allocate() etc
  - **Use:** operates on these resources
  - **Release:** process releases the resources. Eg. Using a system call like release(), close(), signal(), free etc.
- Request and release of semaphore, acquire and release of lock on mutex

Deadlock with physical resources

Example 1

- Consider a system with 3 CD RW drives.
- Suppose 3 processes(p0,p1,p2) are holding one drive each.
- What happens,
  - If a process p0 makes a request for one more drive

Example 2

- Consider a system with one printer one DVD drive.
- Process Pi is holding printer and process Pj is holding DVD drive.
- What happens if
  - Process Pi request DVD and Pj requests printer

Does dead lock occur in example 1 and 2?

Deadlock with Semaphores

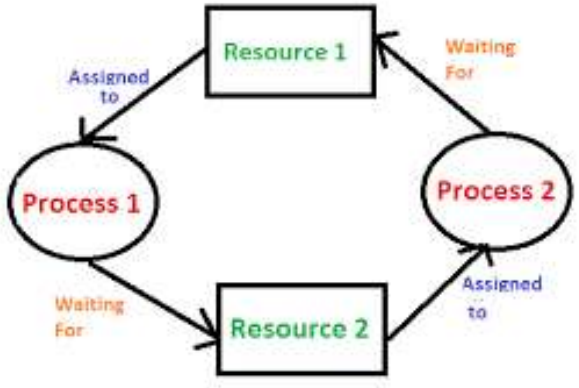
- Data:
  - A semaphore **S1** initialized to 1
  - A semaphore **S2** initialized to 1
- Two processes P1 and P2
- **P1:**

  wait(s1)

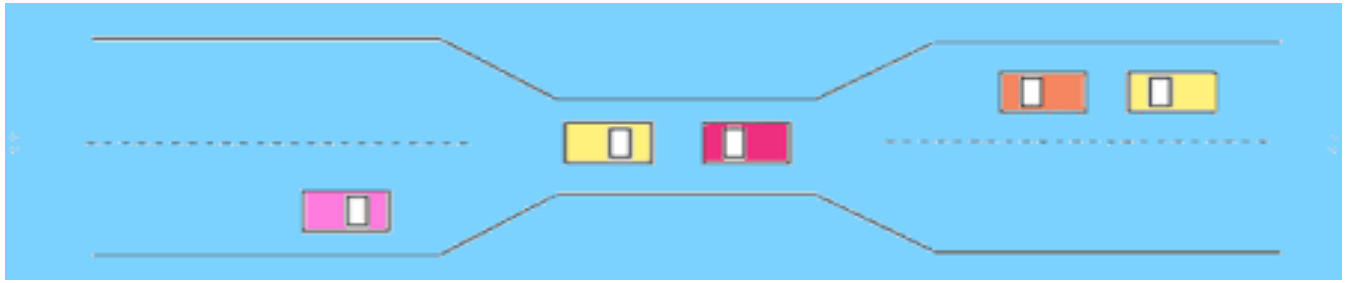
  wait(s2)
- **P2:**

  wait(s2)

  wait(s1)



Bridge crossing Example



- Traffic only in one direction
- Each section of a bridge can be viewed as a resource
- If a deadlock occurs, it can be resolved if one car backs up (preempt resources and rollback)
- Several cars may have to be backed up if a deadlock occurs
- Starvation is possible.
- Note – Most OSes do not prevent or deal with deadlocks

Deadlock Characterization

Deadlock can arise if four conditions hold simultaneously.

- **Mutual exclusion**: only one process at a time can use a resource (sharable resources like Read-only files do not require mutually exclusive access and thus cannot be involved in a deadlock.
- **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.

Resource-Allocation Graph

- Deadlocks are described precisely with directed graphs called system resource-allocation graph

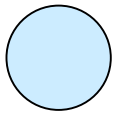
A graph consists of set of vertices V and a set of edges E.

- 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 $R_j \rightarrow P_i$

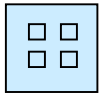
Resource-Allocation Graph (Cont.)

A set of vertices *V* and a set of edges *E*.

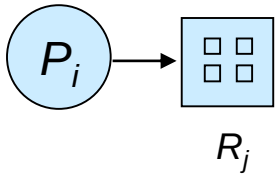
- Process



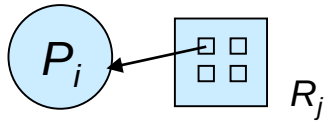
- Resource Type with 4 instances



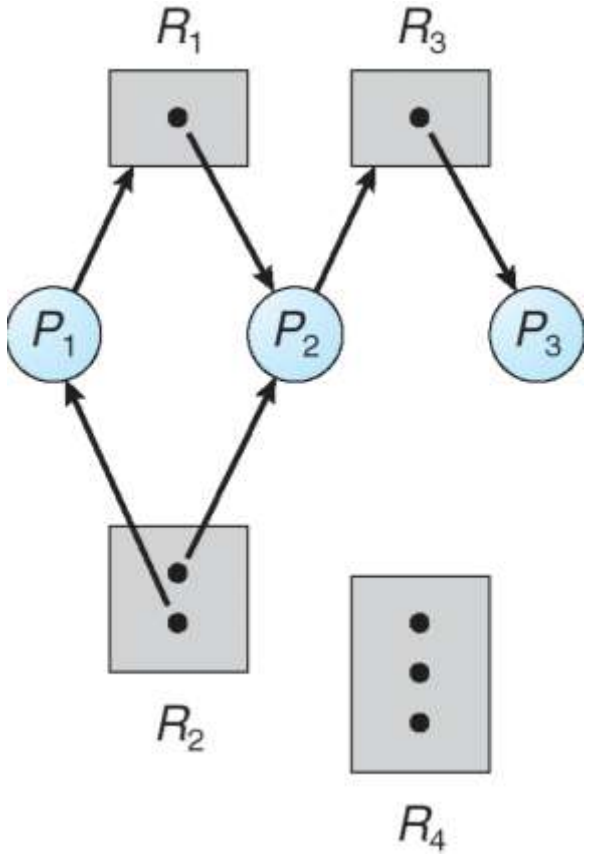
- P_i requests instance of R_j



- P_i is holding an instance of R_j



Example of a Resource-Allocation Graph

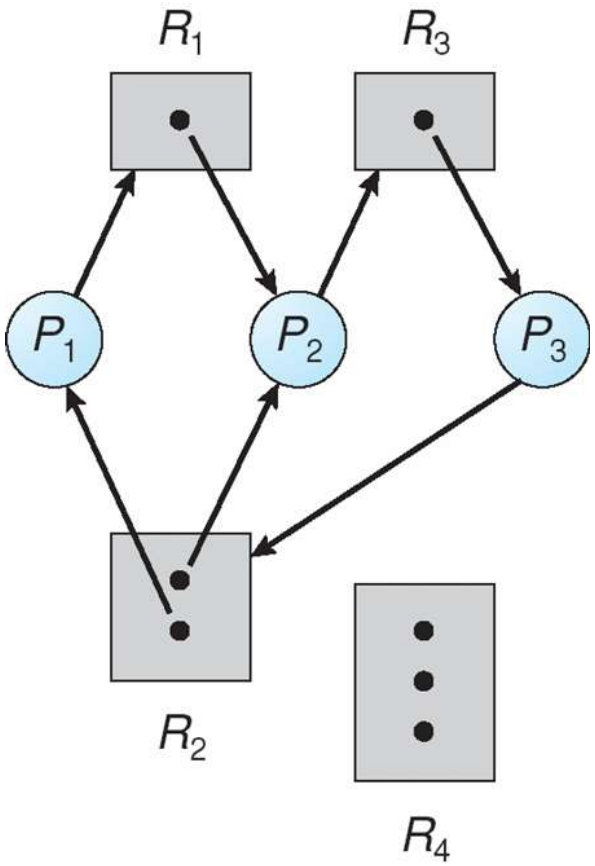


Set of process: P1,P2,P3

Set of Resources: R1,R2,R3

OPERATING SYSTEMS

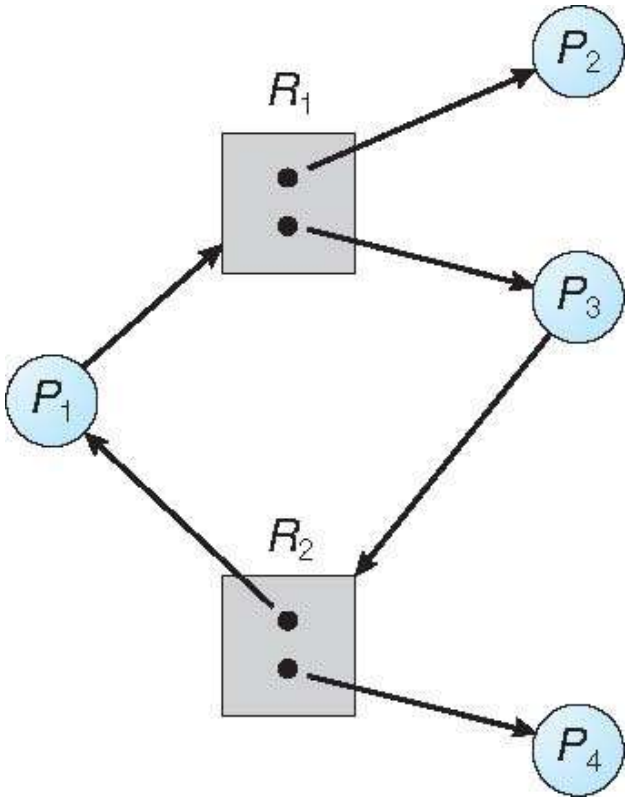
Resource-Allocation Graph With A Deadlock



P1→R1→ P2→R3→ P3→R2→P1

P2→R3→P3→R2→P2

Resource-Allocation Graph With A Cycle but No Deadlock



P1→R1→P3→R2→P1

Basic Facts

- If graph contains no cycles ⇒ no deadlock
- If graph contains a cycle ⇒
  - if only one instance per resource type, then deadlock
  - if several instances per resource type, the system may or may not be in a deadlocked state

THANK YOU

Suresh Jamadagni
Department of Computer Science and Engineering

sureshjamadagni@pes.edu

OPERATING SYSTEMS

Deadlocks

Suresh Jamadagni
Department of Computer Science

OPERATING SYSTEMS

Slides Credits for all the PPTs of this course

- The slides/diagrams in this course are an **adaptation**, **combination**, and **enhancement** of material from the following resources and persons:

1. Slides of Operating System Concepts, Abraham Silberschatz, Peter Baer Galvin, Greg Gagne - 9th edition 2013 and some slides from 10th edition 2018
2. Some conceptual text and diagram from Operating Systems - Internals and Design Principles, William Stallings, 9th edition 2018
3. Some presentation transcripts from A. Frank – P. Weisberg
4. Some conceptual text from Operating Systems: Three Easy Pieces, Remzi Arpaci-Dusseau, Andrea Arpaci Dusseau

OPERATING SYSTEMS

Deadlock Prevention, Deadlock Example

Suresh Jamadagni
Department of Computer Science

Methods for Handling Deadlocks

- Generally speaking there are three ways of handling deadlocks:
  - Deadlock prevention or avoidance - Do not allow the system to get into a deadlocked state.
  - Allow the system to enter into deadlocked state, detect it, and recover.
  - Ignore the problem all together and pretend that deadlocks never occur in the system.

Deadlock Prevention

4 Necessary conditions for deadlock to occur

- **Mutual Exclusion**
  - At least one resource must be non sharable
    - Ex. printers and tape drives, mutex locks
  - Sharable resources do not require mutual exclusion
    - Ex . Read-only files
- **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 (Cont.)

- **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
  - Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting
- **Circular Wait** –
  - Each resource will be assigned with a numerical number.
  - A process can request the resources increasing/decreasing order of numbering.

Circular Wait

- Ex., if P1 process is allocated R5 resources, now next time if P1 ask for R4, R3 lesser than R5 such request will not be granted, only request for resources more than R5 will be granted.
- Resources={R1,R2,…..Rm}
- Resources are assigned unique number
- Each process request resource in increasing order enumeration
- we define a one-to-one function F: R→N, where N is the set of natural numbers
- Protocol 1: Process makes request for Ri and then for Rj. Resources Rj request is allowed if and only if F(Rj)>F(Ri).
- Protocol 2: Process requesting an instance of resource type Rj, must have released any resource Ri , such that F(Ri)>=F(Rj).
- If these protocols are used then circular wait will not exist.

Circular Wait

If these two protocols are used, then the circular-wait condition cannot hold.

Proof by contradiction:

- We can demonstrate this fact that by assuming that circular wait condition cannot hold
- Consider set of processes P={P0,P1,……Pn}
- Let us consider process P0 is waiting for resource held by P1, P1 waiting for P2,……..Pn-1 is waiting resource held by Pn, Pn is waiting for resources held by P0.
- Generalizing this, Process Pi is waiting for resources Ri, Ri is held by Pi+1 and it is making request for Ri+1
- We must have F(Ri)<F(Ri+1)
- But this condition means that $F(R0) < F(R1) < ... < F(Rn) < F(R0)$.
- By transitivity, $F(R0) < F(R0)$, which is impossible
- Therefore no circular wait.

Circular Wait

Example

- F(tape drive)=1
- F(disk drive)=5
- F(printer)=12
- F(tape drive)<F(printer)

Circular Wait

- Ex., if P1 process is allocated R5 resources, now next time if P1 ask for R4, R3 lesser than R5 such request will not be granted, only request for resources more than R5 will be granted.
- Invalidating the circular wait condition is most common.
- Assign each resource (i.e., mutex locks) a unique number.
- Resources must be acquired in order.
- If:
  first_mutex = 1
  second_mutex = 5

  code for **thread_two** could not be written as follows:

OPERATING SYSTEMS

Deadlock Example

```
/* thread one runs in this function */
void *do_work_one(void *param)
{
   pthread_mutex_lock(&first_mutex);
   pthread_mutex_lock(&second_mutex);
   /** * Do some work */
   pthread_mutex_unlock(&second_mutex);
   pthread_mutex_unlock(&first_mutex);
   pthread_exit(0);
}
/* thread two runs in this function */
void *do_work_two(void *param)
{
   pthread_mutex_lock(&second_mutex);
   pthread_mutex_lock(&first_mutex);
   /** * Do some work */
   pthread_mutex_unlock(&first_mutex);
   pthread_mutex_unlock(&second_mutex);
   pthread_exit(0);
}
```

Deadlock Example with Lock Ordering

```
void transaction(Account from, Account to, double amount) {
   mutex lock1, lock2;
   lock1 = get_lock(from);
   lock2 = get_lock(to);
   acquire(lock1);
      acquire(lock2);
         withdraw(from, amount);
         deposit(to, amount);
      release(lock2);
   release(lock1);
}
```

Transactions 1 and 2 execute concurrently. Transaction 1 transfers $25 from account A to account B, and Transaction 2 transfers $50 from account B to account A

OPERATING SYSTEMS

Deadlock Detection, Algorithm

Chandravva Hebbi
Department of Computer Science

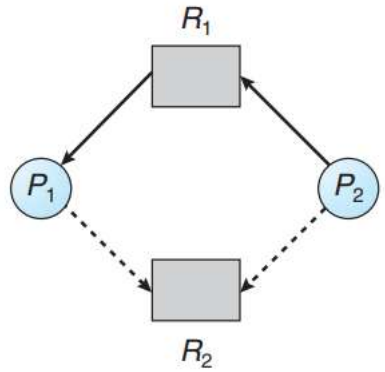
Deadlock Detection

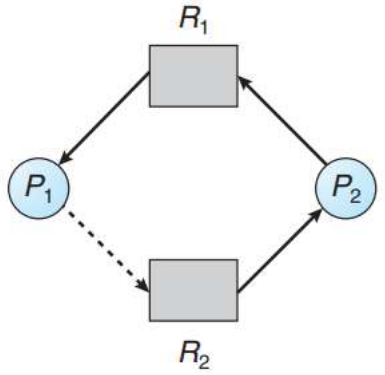
- Allow system to enter deadlock state
- Detection algorithm
- Recovery scheme

OPERATING SYSTEMS

Single Instance of Each Resource Type

- In a resource allocation graph, a claim edge $P_i \rightarrow R_j$ indicates that process P_i may request resource R_j at some time in the future. This edge is represented in the graph by a dashed line.
- 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

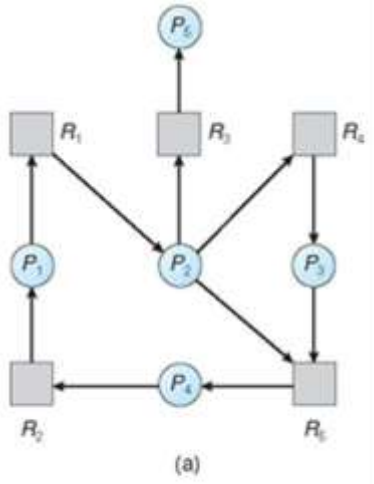




OPERATING SYSTEMS

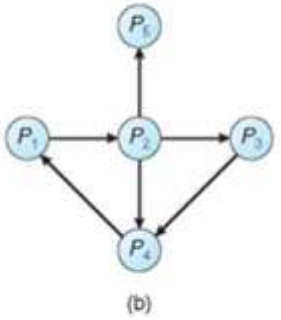
Resource-Allocation Graph and Wait-for Graph

- If all resources have only a single instance, then we can define a deadlock detection algorithm that uses a variant of the resource-allocation graph, called a wait-for graph.
- We obtain wait-for graph from the resource-allocation graph by removing the resource nodes and collapsing the appropriate edges.
- An edge from P_i to P_j implies that process P_i is waiting for process P_j to release a resource that P_i needs.
- An edge $P_i \rightarrow P_j$ exists in a wait-for graph if and only if the corresponding resource allocation graph contains two edges $P_i \rightarrow R_q$ and $R_q \rightarrow P_j$ for some resource R_q
- A **deadlock exists** in the system if and only if the **wait-for graph contains a cycle**.
- 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



(a)

Resource-Allocation Graph



(b)

Corresponding wait-for graph

Several Instances of a Resource Type

- **Available***:* A vector of length ***m*** indicates the number of available resources of each type. If Available[j] = k, then k instances of resource type R_j are available
- **Max**: An n × m matrix defines the maximum demand of each process. If Max[i][j] = k, then process P_i may request at most k instances of resource type R_j .
- **Allocation***:* An ***n*** **x** ***m*** matrix defines the number of resources of each type currently allocated to each process. If Allocation[i][j] = k, then process P_i is currently allocated k instances of resource type R_j
- **Request***:* An ***n*** **x** ***m*** matrix indicates the current request of each process. If ***Request*** **[*i*][*j*] = *k***, then process P_i is requesting ***k*** more instances of resource type R_j.

OPERATING SYSTEMS

Detection Algorithm

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

   (a) ***Work* = *Available***

   (b) For ***i* = 1,2, …, *n***, if $\boldsymbol{Allocation_i \neq 0}$, then
   Finish*[i] = *false; otherwise, ***Finish*[i] = *true***

2. Find an index ***i*** such that both:

   (a) ***Finish*[*i*] == *false***

   (b) $\boldsymbol{Request_i \leq Work}$

   If no such ***i*** exists, go to step 4

Detection Algorithm (Cont.)

3. ***Work = Work + Allocation***$_i$
   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. If **Finish[i] == true** for all i, then the system is in a safe state

Algorithm requires an order of O(m x n^2) operations to detect whether the system is in deadlocked state

OPERATING SYSTEMS

Example of Detection Algorithm

- 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***

Example of Detection Algorithm (Cont.)

- P_2 requests an additional instance of type ***C***

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

- State of system?
  - Can reclaim resources held by process P_0, but insufficient resources to fulfill other processes; requests
  - Deadlock exists, consisting of processes P_1, P_2, P_3, and P_4

OPERATING SYSTEMS

Detection-Algorithm Usage

- When, and how often, to invoke depends on:
  - How often a deadlock is likely to occur?
  - How many processes will need to be rolled back?
    - one for each disjoint cycle
- If detection algorithm is invoked arbitrarily, there may be many cycles in the resource graph and so we would not be able to tell which of the many deadlocked processes “caused” the deadlock.

THANK YOU

Suresh Jamadagni
Department of Computer Science and Engineering

sureshjamadagni@pes.edu

OPERATING SYSTEMS

Deadlocks

Suresh Jamadagni
Department of Computer Science

OPERATING SYSTEMS

Slides Credits for all the PPTs of this course

- The slides/diagrams in this course are an **adaptation**, **combination**, and **enhancement** of material from the following resources and persons:

1. Slides of Operating System Concepts, Abraham Silberschatz, Peter Baer Galvin, Greg Gagne - 9th edition 2013 and some slides from 10th edition 2018
2. Some conceptual text and diagram from Operating Systems - Internals and Design Principles, William Stallings, 9th edition 2018
3. Some presentation transcripts from A. Frank – P. Weisberg
4. Some conceptual text from Operating Systems: Three Easy Pieces, Remzi Arpaci-Dusseau, Andrea Arpaci Dusseau

OPERATING SYSTEMS

Deadlock Avoidance

Suresh Jamadagni
Department of Computer Science

OPERATING SYSTEMS

Deadlock Avoidance

- Deadlock-prevention algorithms, prevent deadlocks by limiting how requests can be made.
- The limits ensure that at least one of the necessary conditions for deadlock cannot occur.
- Possible side effects of preventing deadlocks by this method, however, are low device utilization and reduced system throughput.
- **Deadlock avoidance** requires that the operating system be given additional information in advance concerning which resources a process will request and use during its lifetime.
- With this additional knowledge of complete sequence of requests and releases for each process, the system can decide for each request whether or not the process should wait in order to avoid a possible future deadlock
- To decide whether the current request can be satisfied or must be delayed, the system must consider the resources currently available, the resources currently allocated to each process, and the future requests and releases of each process

OPERATING SYSTEMS

Deadlock Avoidance

- The simplest and most useful model requires that each process declare the ***maximum number*** of resources of each type that it may need.
- Given this apriori information, it is possible to construct an algorithm that ensures that the system will never enter a deadlocked state.
- A deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that a circular-wait condition can never exist.
- The resource allocation ***state*** is defined by the number of available and allocated resources and the maximum demands of the processes.

OPERATING SYSTEMS

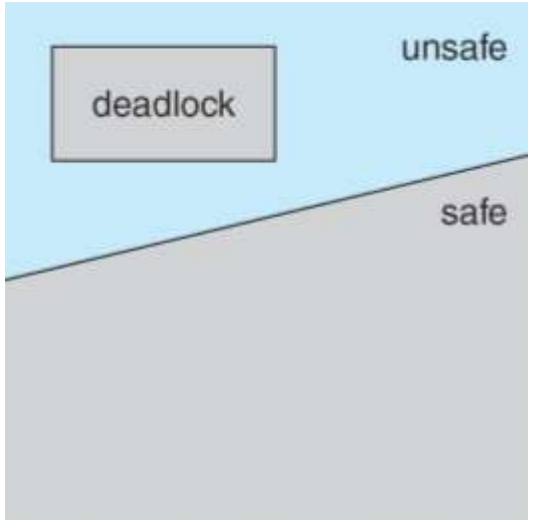
Safe State

- A state is ***safe*** if the system can allocate resources to each process (up to its maximum) in some order and still avoid a deadlock.
- More formally, a system is in a safe state only if there exists a **safe sequence**.
- A sequence of processes <*P*1, *P*2, ..., *Pn*> is a safe sequence for the current allocation state if, for each *Pi* , the resource requests that *Pi* can still make can be satisfied by the currently available resources plus the resources held by all *Pj*, with *j < i.*
- In this situation, if the resources that *Pi* needs are not immediately available, then *Pi* can wait until all *Pj* have finished.
- When they have finished, *Pi* can obtain all of its needed resources, complete its designated task, return its allocated resources, and terminate.
- When *Pi* terminates, *Pi*+1 can obtain its needed resources, and so on. If no such sequence exists, then the system state is said to be ***unsafe***

OPERATING SYSTEMS

Safe, Unsafe and Deadlock States

- A safe state is not a deadlocked state.
- Conversely, a deadlock state is an unsafe state.
- Not all unsafe states are deadlocks.
- An unsafe state may lead to a deadlock.
- As long as the state is safe, the OS can avoid unsafe states.
- In an unsafe state, the OS cannot prevent processes from requesting resources in such a way that a deadlock occurs.
- The behavior of processes controls unsafe states.
- If a system is in safe state → no deadlocks
- If a system is in unsafe state → possibility of deadlock
- Goal for Avoidance → ensure that a system will never enter an unsafe state.



OPERATING SYSTEMS

Example - Safe, Unsafe and Deadlock States

- Consider a system with twelve magnetic tape drives and three processes: *P*0, *P*1, and *P*2.
- Process *P*0 requires ten tape drives, process *P*1 may need as many as four tape drives, and process *P*2 may need up to nine tape drives.

	Maximum Needs	Current Needs
P_0	10	5
P_1	4	2
P_2	9	2

- Suppose that, at time *t*0, process *P*0 is holding five tape drives, process *P*1 is holding two tape drives, and process *P*2 is holding two tape drives. (Thus, there are three free tape drives.)
- At time *t*0, the system is in a safe state. The sequence <*P*1, *P*0, *P*2> satisfies the safety condition. Process *P*1 can immediately be allocated all its tape drives and then return them (the system will then have five available tape drives); then process *P*0 can get all its tape drives and return them (the system will then have ten available tape drives); and finally process *P*2 can get all its tape drives and return them
- At time *t*1, process *P*2 requests and is allocated one more tape drive. The system is no longer in a safe state.
- Since process *P*0 is allocated five tape drives but has a maximum of ten, it may request five more tape drives. If it does so, it will have to wait, because they are unavailable. Similarly, process *P*2 may request six additional tape drives and have to wait, resulting in a deadlock

OPERATING SYSTEMS

Deadlock avoidance algorithms

- Avoidance algorithms ensure that the system will never deadlock.
- The idea is simply to ensure that the system will always remain in a safe state.
- Initially, the system is in a safe state. Whenever a process requests a resource that is currently available, the system must decide whether the resource can be allocated immediately or whether the process must wait.
- The request is granted only if the allocation leaves the system in a safe state.
- In this scheme, if a process requests a resource that is currently available, it may still have to wait.
- Thus, resource utilization may not be optimal

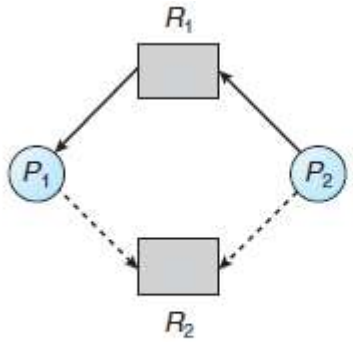
OPERATING SYSTEMS

Resource Allocation Graph algorithm

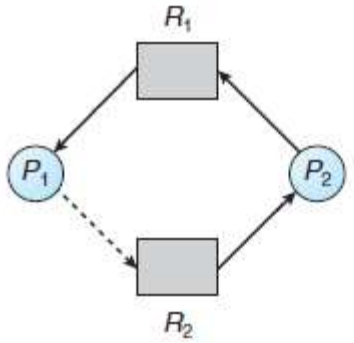
- In addition to the request and assignment edges in a resource allocation graph, we introduce a new type of edge, called a **claim edge**
- A claim edge *Pi* → *Rj* indicates that process *Pi* may request resource *Rj* at some time in the future. This edge resembles a request edge in direction but is represented in the graph by a dashed line.
- When process *Pi* requests resource *Rj* , the claim edge *Pi* → *Rj* is converted to a request edge. Similarly, when a resource *Rj* is released by *Pi* , the assignment edge *Rj* → *Pi* is reconverted to a claim edge *Pi* → *Rj*
- The resources must be claimed a priori in the system. That is, before process *Pi* starts executing, all its claim edges must already appear in the resource-allocation graph.
- Now suppose that process *Pi* requests resource *Rj*. The request can be granted only if converting the request edge *Pi* → *Rj* to an assignment edge *Rj* → *Pi* does not result in the formation of a cycle in the resource-allocation graph.

OPERATING SYSTEMS

Resource Allocation Graph algorithm example



Resource-allocation graph for deadlock avoidance.



An unsafe state in a resource-allocation graph.

- Consider the above resource-allocation graph.
- Suppose that *P*2 requests *R*2. Although *R*2 is currently free, we cannot allocate it to *P*2, since this action will create a cycle in the graph below.
- A cycle indicates that the system is in an unsafe state.
- If *P*1 requests *R*2, then a deadlock will occur.

OPERATING SYSTEMS

Banker's algorithm

- When a new process enters the system, it must declare the maximum number of instances of each resource type that it may need. This number may not exceed the total number of resources in the system.
- When a user requests a set of resources, the system must determine whether the allocation of these resources will leave the system in a safe state. If it will, the resources are allocated; otherwise, the process must wait until some other process releases enough resources.
- Several data structures must be maintained to implement the banker's algorithm. These data structures encode the state of the resource-allocation system.
- We need the following data structures, where *n* is the number of processes in the system and *m* is the number of resource types

OPERATING SYSTEMS

Banker's algorithm data structures

- **Available –** A vector of length *m* indicates the number of available resources of each type. If ***Available***[*j*] equals *k,* then *k* instances of resource type *Rj* are available.
- **Max**. - An *n* × *m* matrix defines the maximum demand of each process. If ***Max***[*i*][*j*] equals *k,* then process *Pi* may request at most *k* instances of resource type *Rj* .
- **Allocation** - An *n* × *m* matrix defines the number of resources of each type currently allocated to each process. If ***Allocation***[*i*][*j*] equals *k,* then process *Pi* is currently allocated *k* instances of resource type *Rj* .
- **Need** - An *n* × *m* matrix indicates the remaining resource need of each process. If ***Need***[*i*][*j*] equals *k,* then process *Pi* may need *k* more instances of resource type *Rj* to complete its task.
- ***Need***[*i*][*j*] = ***Max***[*i*][*j*] − ***Allocation***[*i*][*j*].
- Treat each row in the matrices ***Allocation*** and ***Need*** as vectorsThe vector ***Allocation****i* specifies the resources currently allocated to process *Pi* ; the vector ***Need****i* specifies the additional resources that process *Pi* may still request to complete its task
- These data structures vary over time in both size and value.

OPERATING SYSTEMS

Banker's algorithm for determining whether or not a system is in a safe state

1. Let***Work*** and ***Finish*** be vectors of length *m* and *n,* respectively.

   Initialize ***Work*** = ***Available*** and ***Finish***[*i*] = ***false*** for *i* = 0, 1, ..., *n* − 1.

2. Find an index *i* such that both

   a. ***Finish***[*i*] == ***false***

   b. ***Need****i* ≤***Work***

   If no such *i* exists, go to step 4

3. ***Work*** =***Work*** + ***Allocation****i*

Finish[*i*] = ***true***

Go to step 2.

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

- This algorithm may require an order of $m \times n^2$ operations to determine whether a state is safe.

OPERATING SYSTEMS

Banker's algorithm for determining whether requests can be safely granted

Let ***Request****i* be the request vector for process *Pi* . If ***Request****i* [*j*] == *k*, then process *Pi* wants *k* instances of resource type *Rj* .

When a request for resources is made by process *Pi* , the following actions are taken:

1. If ***Request****i* ≤ ***Need****i* , go to step 2. Otherwise, raise an error condition, since the process has exceeded its maximum claim.
2. If ***Request****i* ≤ ***Available,*** go to step 3. Otherwise, *Pi* must wait, since the resources are not available.
3. Have the system pretend to have allocated the requested resources to process *Pi* by modifying the state as follows:

Available = ***Available–Request****i* ;

Allocation*i* = ***Allocation****i* + ***Request****i* ;

Need*i* = ***Need****i* –***Request****i* ;

If the resulting resource-allocation state is safe, the transaction is completed, and process *Pi* is allocated its resources.

If the new state is unsafe, then *Pi* must wait for ***Request****i* **,** and the old resource-allocation state is restored.

OPERATING SYSTEMS

Banker's algorithm example

- Consider a system with five processes *P*0 through *P*4 and three resource types *A, B,* and *C.*
- Resource type *A* has ten instances, resource type *B* has five instances, and resource type *C* has seven instances.
- Suppose that, at time *T*0, the following snapshot of the system has been taken:

	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	

- Sequence <*P*1, *P*3, *P*4, *P*0, *P*2> satisfies the safety requirement. Hence, we can immediately grant the request of process *P*1.
- A request for (3,3,0) by *P*4 cannot be granted, since the resources are not available.
- A request for (0,2,0) by *P*0 cannot be granted, even though the resources are available, since the resulting state is unsafe.

THANK YOU

Suresh Jamadagni
Department of Computer Science and Engineering

sureshjamadagni@pes.edu

OPERATING SYSTEMS

Signals

Suresh Jamadagni
Department of Computer Science

OPERATING SYSTEMS

Slides Credits for all the PPTs of this course

- The slides/diagrams in this course are an **adaptation**, **combination**, and **enhancement** of material from the following resources and persons:

1. Slides of Operating System Concepts, Abraham Silberschatz, Peter Baer Galvin, Greg Gagne - 9th edition 2013 and some slides from 10th edition 2018
2. Some conceptual text and diagram from Operating Systems - Internals and Design Principles, William Stallings, 9th edition 2018
3. Some presentation transcripts from A. Frank – P. Weisberg
4. Some conceptual text from Operating Systems: Three Easy Pieces, Remzi Arpaci-Dusseau, Andrea Arpaci Dusseau

OPERATING SYSTEMS

Signals

Suresh Jamadagni
Department of Computer Science

OPERATING SYSTEMS

Signals

- A signal is used in UNIX systems to notify a process that a particular event has occurred.
- A signal may be received either synchronously or asynchronously depending on the source of and the reason for the event being signaled
- All signals, whether synchronous or asynchronous, follow the same pattern:
  1. A signal is generated by the occurrence of a particular event.
  2. The signal is delivered to a process.
  3. Once delivered, the signal must be handled.

OPERATING SYSTEMS

Signals – synchronous and asynchronous

- Examples of synchronous signal include illegal memory access and division by 0.
- If a running program performs either of these actions, a signal is generated.
- Synchronous signals are delivered to the same process that performed the operation that caused the signal (that is the reason they are considered synchronous).
- When a signal is generated by an event external to a running process, that process receives the signal asynchronously.
- Examples of such signals include terminating a process with specific keystrokes (such as <control><C>) and having a timer expire.
- Typically, an asynchronous signal is sent to another process.

Signal handlers

- A signal may be ***handled*** by one of two possible handlers:
  1. A default signal handler
  2. A user-defined signal handler
- Every signal has a **default signal handler** that the kernel runs when handling that signal.
- This default action can be overridden by a **user-defined signal handler** that is called to handle the signal.
- Signals are handled in different ways. Some signals (such as changing the size of a window) are simply ignored; others (such as an illegal memory access) are handled by terminating the program.

Signal handling

- Handling signals in single-threaded programs is straightforward: signals are always delivered to a process.
- Delivering signals is more complicated in multithreaded programs, where a process may have several threads.
- In general, the following options exist:
  1. Deliver the signal to the thread to which the signal applies.
  2. Deliver the signal to every thread in the process.
  3. Deliver the signal to certain threads in the process.
  4. Assign a specific thread to receive all signals for the process.

Signal handling

- The method for delivering a signal depends on the type of signal generated.
- For example, synchronous signals need to be delivered to the thread causing the signal and not to other threads in the process.
- However, the situation with asynchronous signals is not as clear. Some asynchronous signals—such as a signal that terminates a process (<control><C>) should be sent to all threads.
- The standard UNIX function for delivering a signal is

  kill(pid t pid, int signal)
- This function specifies the process (pid) to which a particular signal (signal) is to be delivered

Signal handling

- Most multithreaded versions of UNIX allow a thread to specify which signals it will accept and which it will block.
- An asynchronous signal may be delivered only to those threads that are not blocking it.
- Because signals need to be handled only once, a signal is typically delivered only to the first thread found that is not blocking it.
- POSIX Pthreads provides the following function, which allows a signal to be delivered to a specified thread (tid):

  pthread kill(pthread t tid, int signal)

OPERATING SYSTEMS

Signals in windows

- Windows does not explicitly provide support for signals, it allows to emulate them using **asynchronous procedure calls (APCs)**.
- The APC facility enables a user thread to specify a function that is to be called when the user thread receives notification of a particular event.
- An APC is roughly equivalent to an asynchronous signal in UNIX
- The APC facility is more straightforward, since an APC is delivered to a particular thread rather than a process

OPERATING SYSTEMS

Signals in Linux

- A signal is a kind of software interrupt, used to announce asynchronous events to a process
- **SIGINT** is a signal generated when a user presses Control-C. This will terminate the program from the terminal.
- **SIGALRM** is generated when the timer set by the alarm function goes off.
- **SIGABRT** is generated when a process executes the abort function.
- **SIGSTOP** tells LINUX to pause a process to be resumed later.
- **SIGCONT** tells LINUX to resume the processed paused earlier.
- **SIGSEGV** is sent to a process when it has a segmentation fault.
- **SIGKILL** is sent to a process to cause it to terminate at once.

THANK YOU

Suresh Jamadagni
Department of Computer Science and Engineering

sureshjamadagni@pes.edu