Introduction and Overview

ECE 3574 Applied Software Design

Dr. Roberto Palmieri
ECE Department
Virginia Tech

Textbooks

- Required:
  - An Introduction to Design Patterns in C++ with Qt, Second Edition, Alan Ezust and Paul Ezust, ISBN-10: 0-132-82645-3, ISBN-13: 978-0-132-82645-7, Prentice Hall, 2012
  - Book has a homepage at: https://www.ics.com/design-patterns#.U-4ouLxdVXO
    - Site may require free registration
    - Find links to HTML version of the book, lecture slides, source codes of examples in book, bug reporting system, etc.
  - VT library has the online version of the book at: http://proquest.safaribooksonline.com.ezproxy.lib.vt.edu/9780132851619

Textbooks (contd.)

- Recommended:
  - Operating System Concepts, Ninth Edition, Avi Silberschatz, Peter Baer Galvin, and Greg Gagne, ISBN 978-1-118-06333-0, John Wiley & Sons, Inc., 2012
    - http://www.os-book.com/
  - VT library has the online version of the book at (8th edition): http://proquest.safaribooksonline.com.ezproxy.lib.vt.edu/book/operating-systems-and-server-administration/9780470128725
  - Great OS book!

Outline

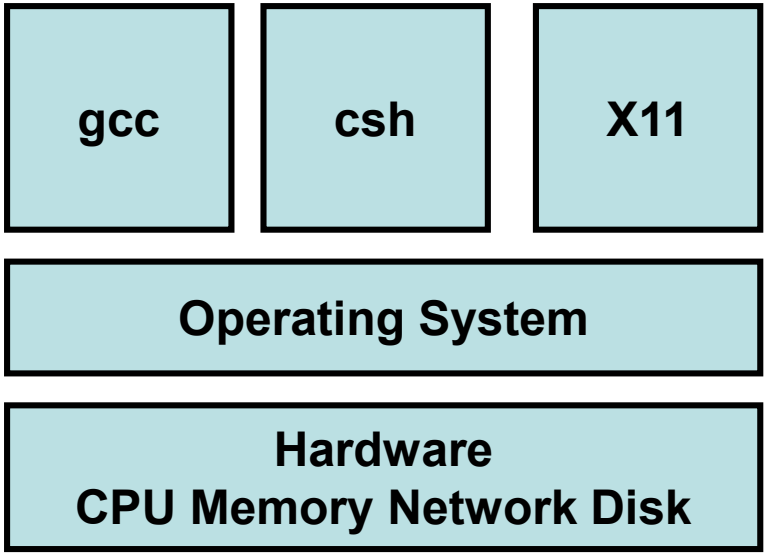
- Introduction to Operating System (NOT IN THE BOOK)
- Introduction to processes (Ch 3, Silberschatz)
- Inter-process communication: shared memory, message passing (Ch 3, Silberschatz)
- Introduction to threads (Ch 4, Silberschatz)

- Process synchronization
  - critical section problem (Ch 6, Silberschatz)
  - software solutions including Peterson's algorithm and generalizations (Ch 6, Silberschatz, Notes)
  - hardware solutions including TAS and CAS-based (Ch 6, Silberschatz, Notes)
  - semaphores (Ch 6, Silberschatz, Notes)
  - classical synchronization problems (Ch 6, Silberschatz, Notes)
  - condition synchronization (Ch 6, Silberschatz, Notes)

- Qt Concurrency (Ch 17, Ezust)

What is an Operating System

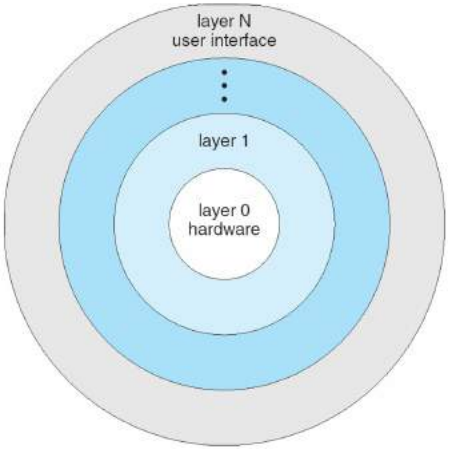
- Software layer that sits between applications and hardware
- Performs services:
  - Abstracts hardware
  - Provides protection
  - Manages resources
- Abstraction is fundamental!!!



OS vs Kernel

- Kernel:
  - the special piece of software that runs with special privileges and actually controls the machine.
- Operating System:
  - Includes also system programs, system libraries, servers, shells, GUI etc.
    - Example: is the Terminal really needed to run your operating system?

What does ”Privileged Mode” mean?

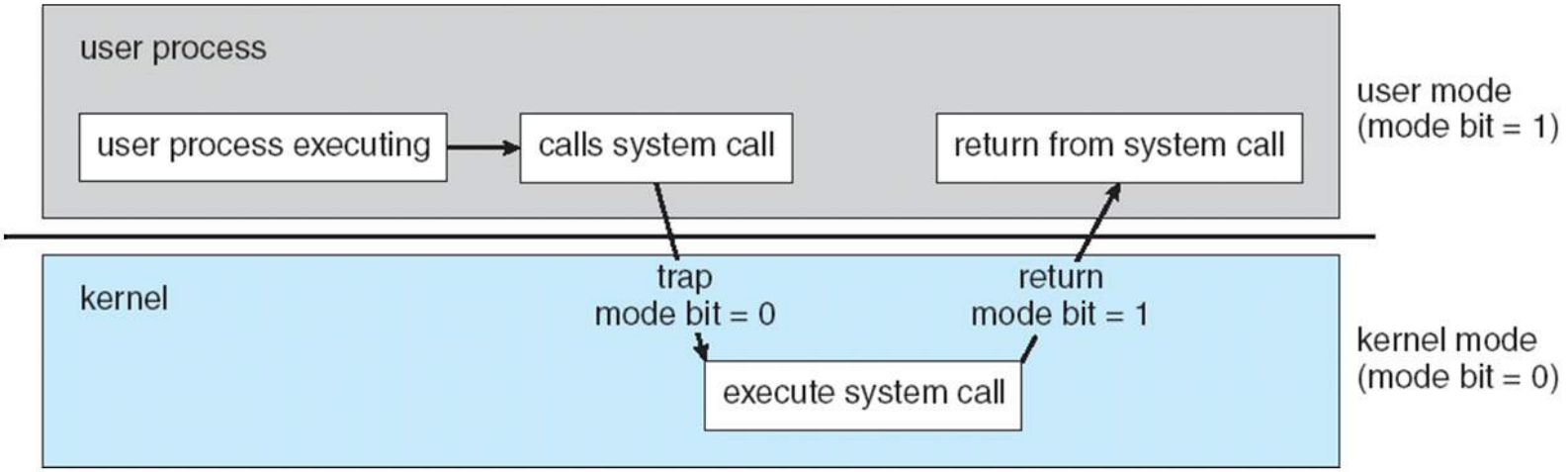


- Two fundamental modes:
  - kernel or privileged mode
  - user or non-privileged mode
- Protection operations can only be performed in kernel mode.
  - Example: HLT is an assembly language instruction that halts the central processing unit (CPU). Pretty much your CPU is gone!

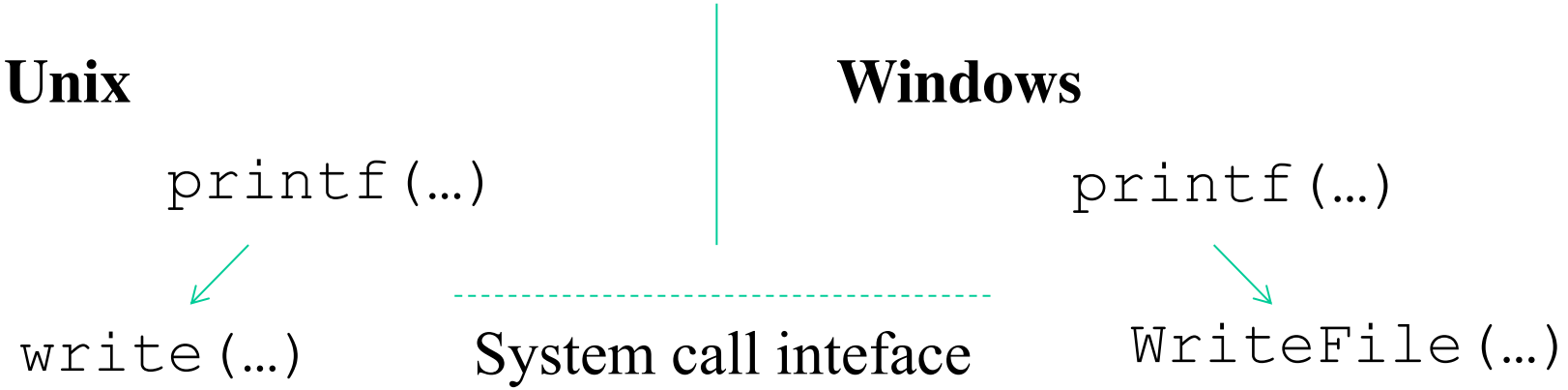
```
int main() {
        ...
        asm(“hlt“)
        ...
}
```

System Calls

- System Calls are functions that represent the gateway to access the operating system's services
- Most high-level instructions produce <u>many</u>



System call is machine dependent



How can the OS manage multiple programs running in parallel?

- Interactive Time-Sharing:
  - Ability of running multiple programs in parallel on a single processor unit
  - Each program receives a slice (*quantum*) of time to execute. To support that:
    - Preemption
    - Scheduler
    - Technique to save and restore execution contexts

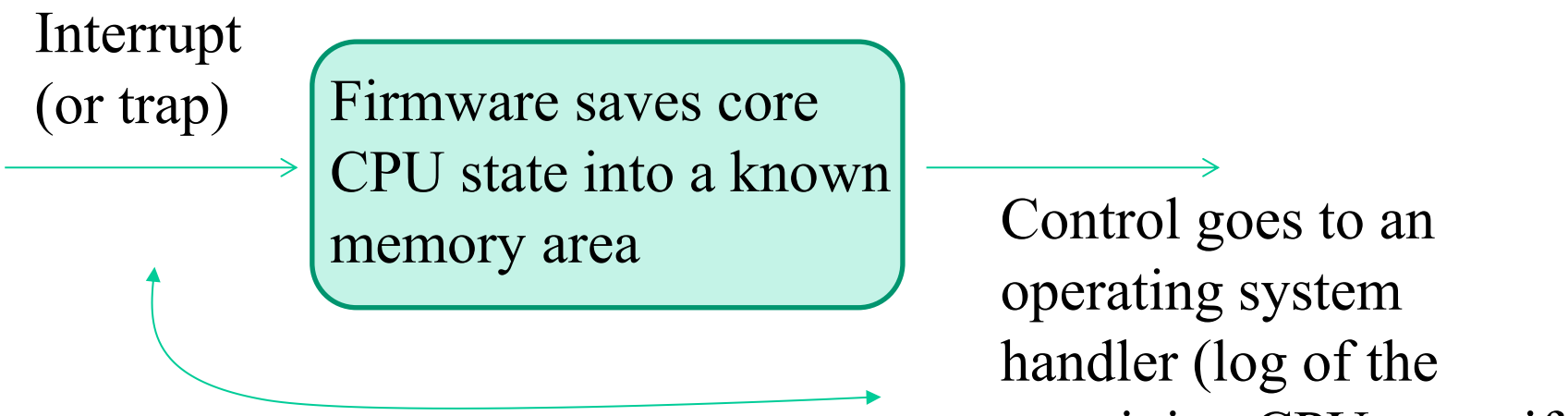
Preemption

- Preemption is a mechanism to reclaim the execution of a program while it is executing
- Preemption can happen when (examples):
  - An interrupt is received, so that the OS can handle it
  - The time slice assigned to a program to execute expires
  - …

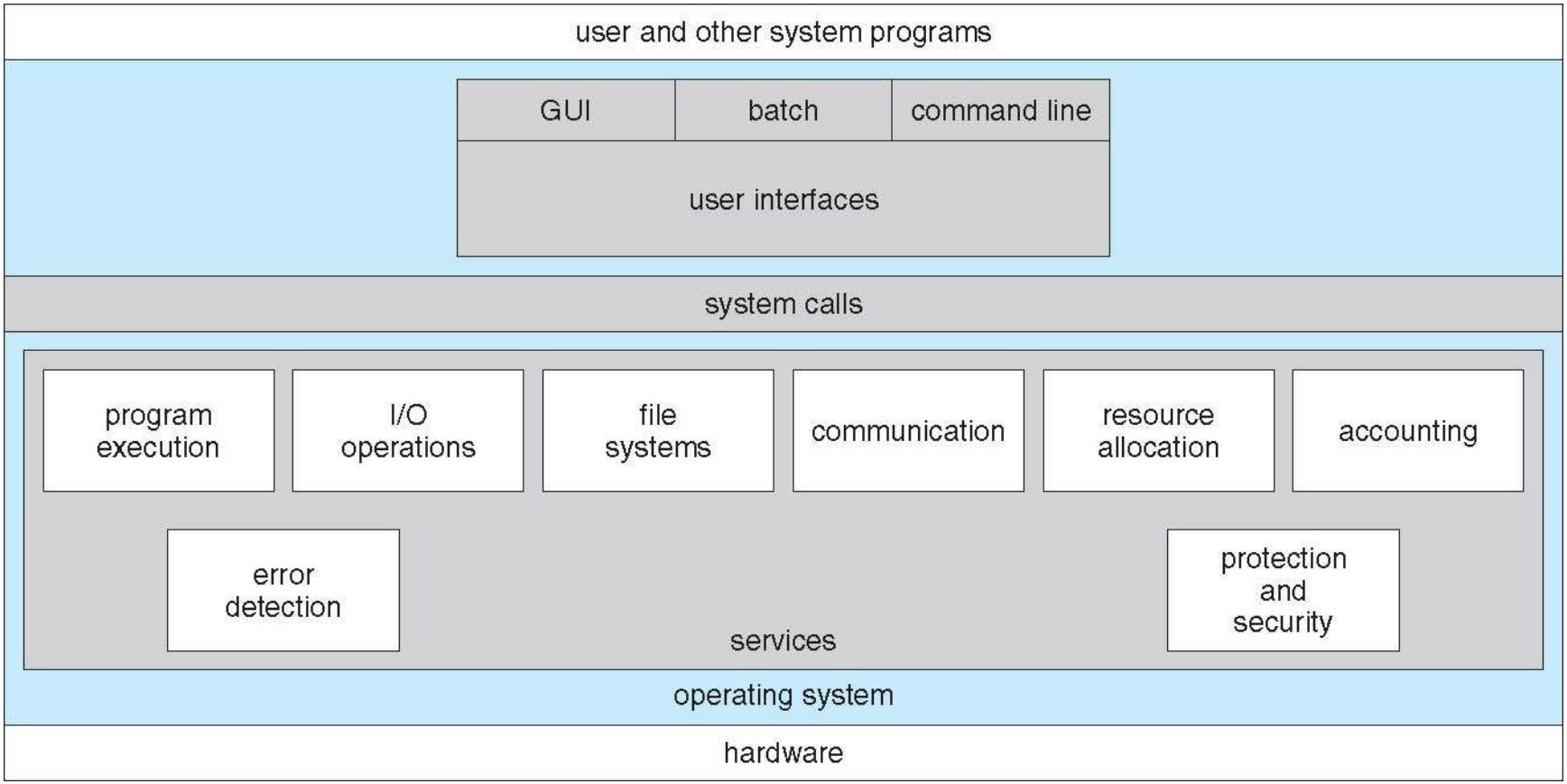
Interrupt Driven

- Modern operating systems are interrupt driven
  - Interrupt is a software or hardware notification that some event happened. Examples:
    - Mouse click
    - Time slice is over
    - Reset button pressed
  - A routine is activated once an interrupt is received, the driver

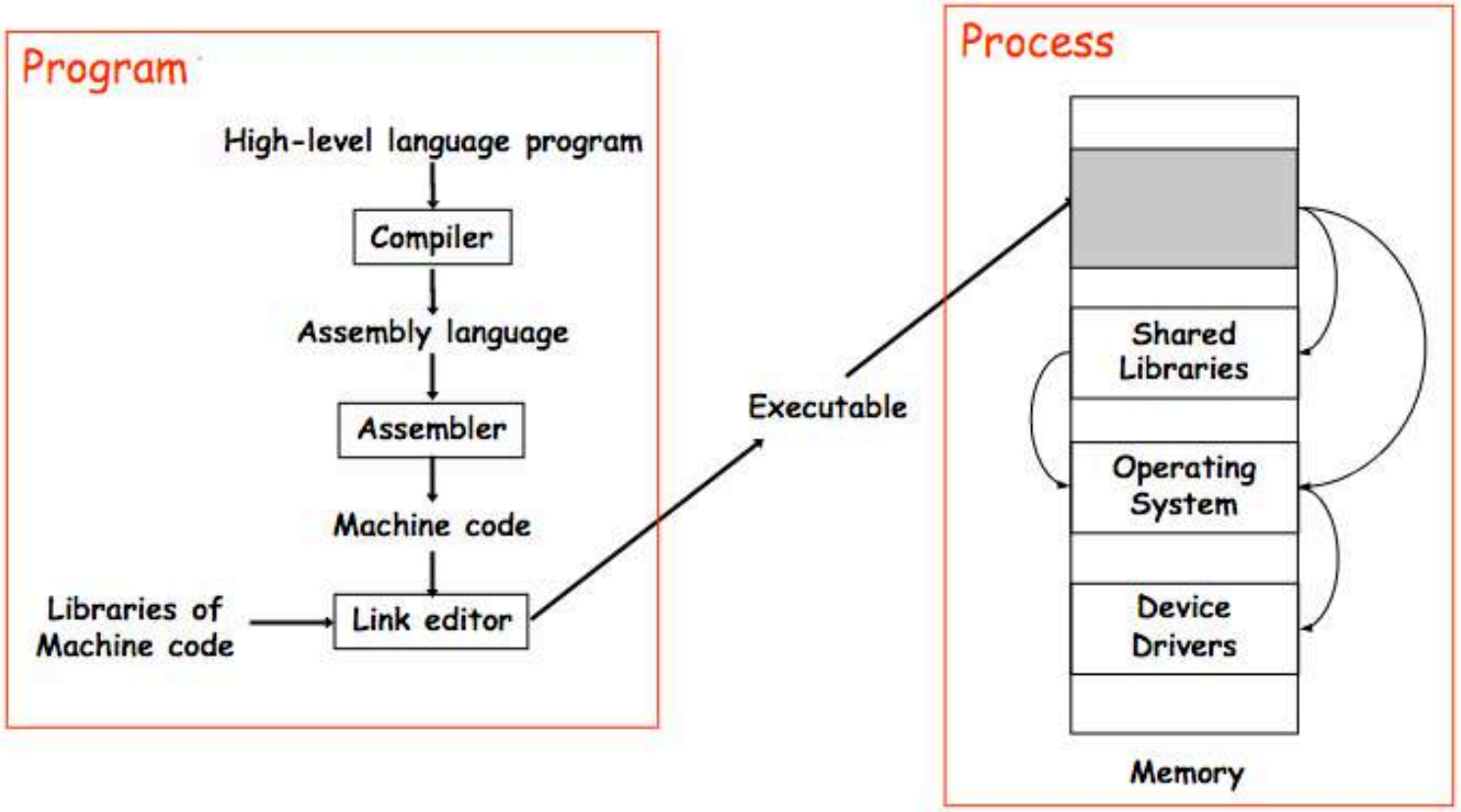
Interrupt driven example



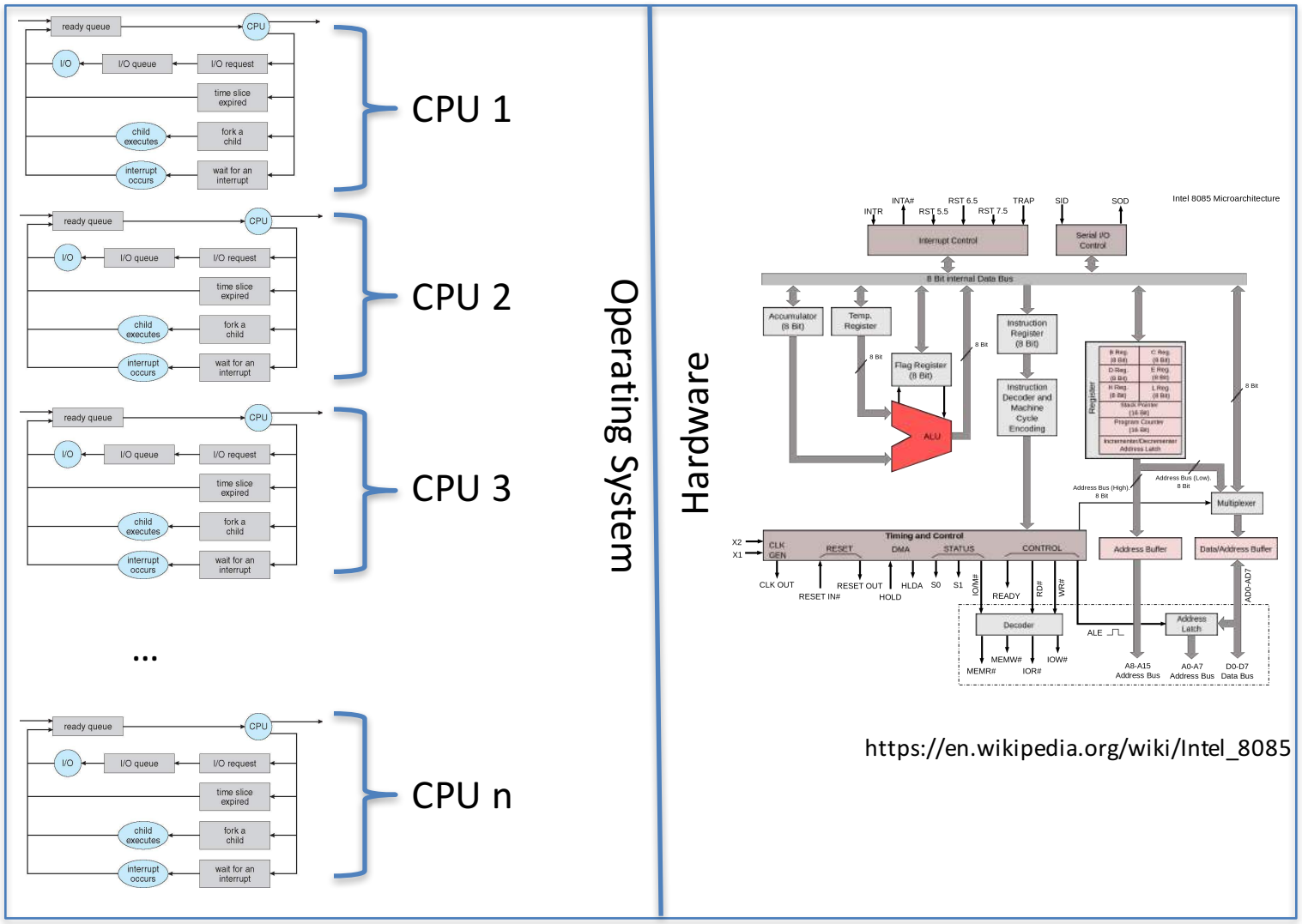
General Architecture OS



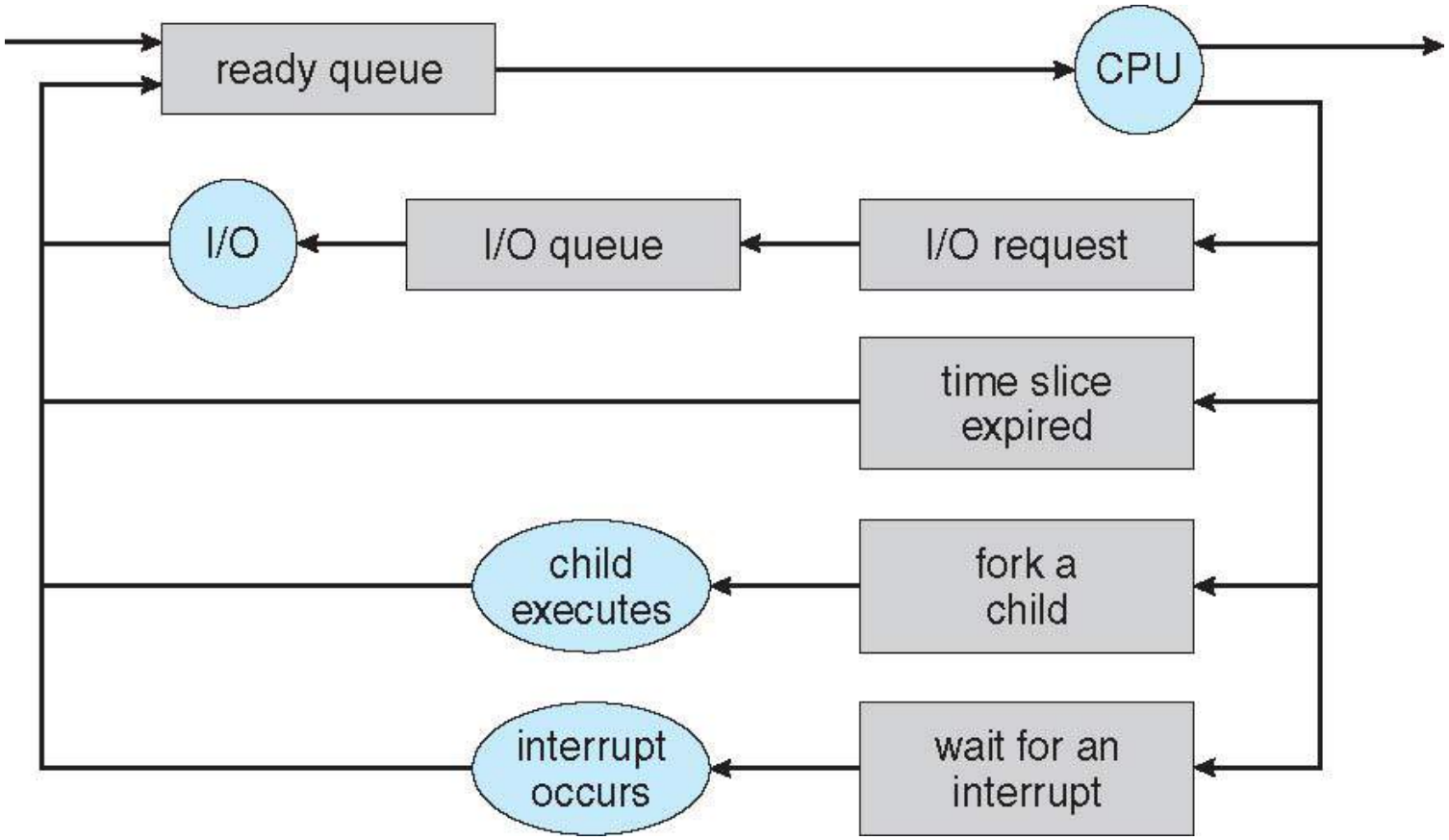
How does a program execute in the OS?



Software and Hardware interaction



Zoom In


ready queue
CPU
I/O
I/O queue
I/O request
time slice expired
child executes
fork a child
interrupt occurs
wait for an interrupt

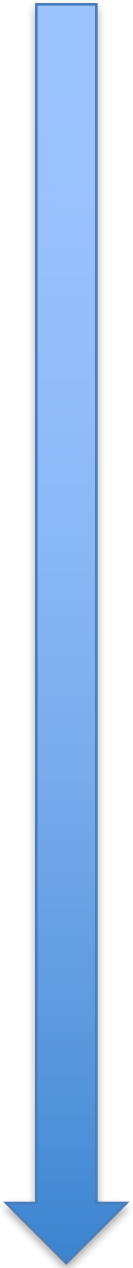
How the program executes…so far?

- You assumed so far that your code executes sequentially, from top to bottom.

```
int main(int argc, char *argv[])
{
    QCoreApplication a(argc, argv);
    QFile textFile("playlist1.tsv");
    QFile binaryFile("playlist1.bin");
    QTextStream textStream;
    QDataStream dataStream;

    if (textFile.open(QIODevice::ReadOnly)) {
        textStream.setDevice(&textFile);
        qtCout << "open text file for mdv" << endl;
    }
    else
        return EXIT_FAILURE;
    if (binaryFile.open(QIODevice::WriteOnly))  {
        dataStream.setDevice(&binaryFile);
        qtCout << "open binary file for mdv" << endl;
    }
    else
        return EXIT_FAILURE;

        ……..……
```

1st

2nd



What about they execute in parallel?

```
int main(int argc, char *argv[])
{
    QCoreApplication a(argc, argv);
    QFile textFile("playlist1.tsv");
    QFile binaryFile("playlist1.bin");
    QTextStream textStream;
    QDataStream dataStream;

    if (textFile.open(QIODevice::ReadOnly)) {
        textStream.setDevice(&textFile);
        qtCout << "open text file for mdv" << endl;
    }
    else
        return EXIT_FAILURE;
    if (binaryFile.open(QIODevice::WriteOnly)) {
        dataStream.setDevice(&binaryFile);
        qtCout << "open binary file for mdv" << endl;
    }
    else
        return EXIT_FAILURE;

        ..............
```

```
int main(int argc, char *argv[])
{
    QCoreApplication a(argc, argv);
    QFile textFile("playlist1.tsv");
    QFile binaryFile("playlist1.bin");
    QTextStream textStream;
    QDataStream dataStream;

    if (textFile.open(QIODevice::ReadOnly)) {
        textStream.setDevice(&textFile);
        qtCout << "open text file for mdv" << endl;
    }
    else
        return EXIT_FAILURE;
    if (binaryFile.open(QIODevice::WriteOnly)) {
        dataStream.setDevice(&binaryFile);
        qtCout << "open binary file for mdv" << endl;
    }
    else
        return EXIT_FAILURE;

        ..............
```

What about they execute in parallel?

```
int main(int argc, char *argv[])
{
    QCoreApplication a(argc, argv);
    QFile textFile("playlist1.tsv");
    QFile binaryFile("playlist1.bin");
    QTextStream textStream;
    QDataStream dataStream;

    if (textFile.open(QIODevice::ReadOnly)) {
        textStream.setDevice(&textFile);
        qtCout << "open text file for mdv" << endl;
    }
    else
        return EXIT_FAILURE;
    if (binaryFile.open(QIODevice::WriteOnly)) {
        dataStream.setDevice(&binaryFile);
        qtCout << "open binary file for mdv" << endl;
    }
    else
        return EXIT_FAILURE;

        ..............
```

```
int main(int argc, char *argv[])
{
    QCoreApplication a(argc, argv);
    QFile textFile("playlist1.tsv");
    QFile binaryFile("playlist1.bin");
    QTextStream textStream;
    QDataStream dataStream;

    if (textFile.open(QIODevice::ReadOnly)) {
        textStream.setDevice(&textFile);
        qtCout << "open text file for mdv" << endl;
    }
    else
        return EXIT_FAILURE;
    if (binaryFile.open(QIODevice::WriteOnly)) {
        dataStream.setDevice(&binaryFile);
        qtCout << "open binary file for mdv" << endl;
    }

                FAILURE;

            ..............
```

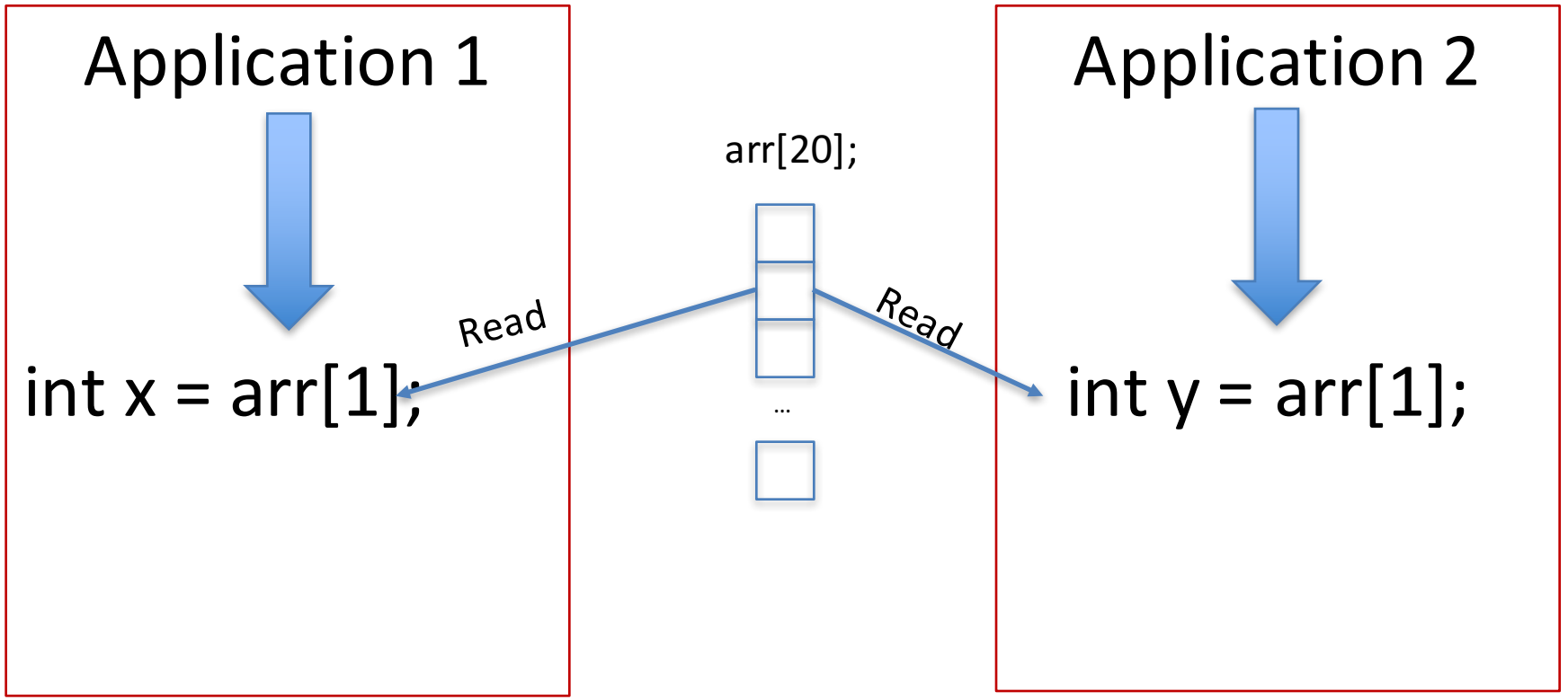
CONFLICT!!

Why do we need parallelism?

- To improve performance (how fast the application can go)
- Ideal example:
  - Application execution time is 10s when executes on one CPU (Sequentially)
  - If it executes in parallel:
    - 5s on two CPUs
    - 2.5s on four CPUs
    - 1.25s on 8 CPUs
    - ….

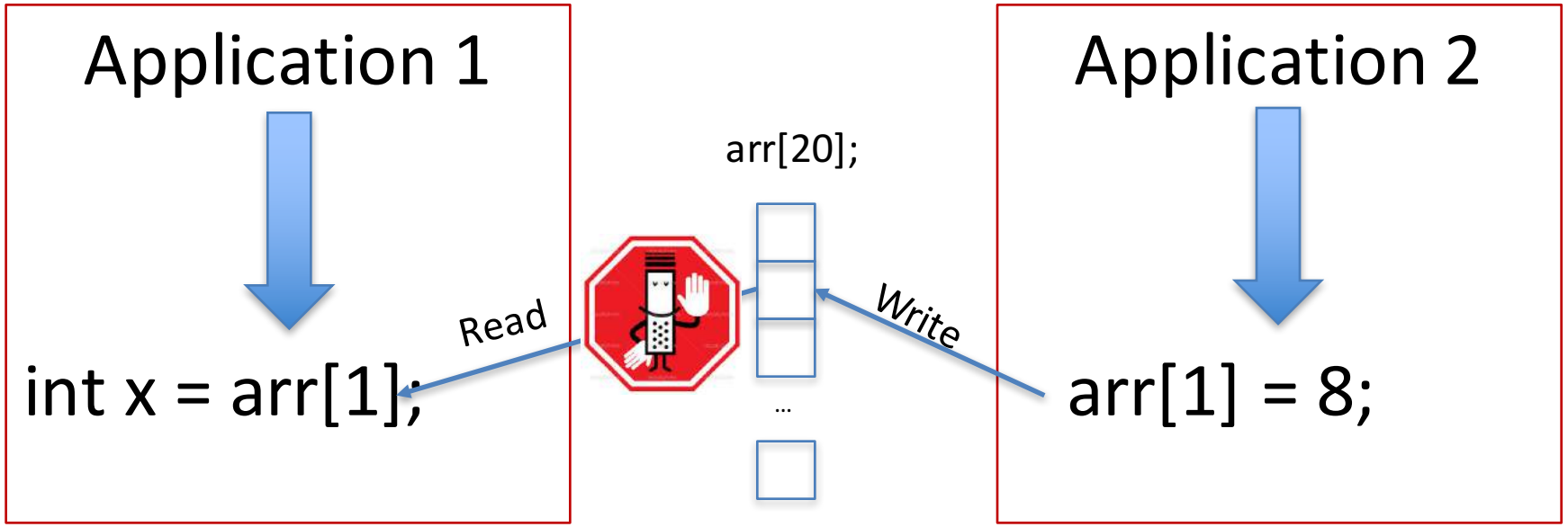
With or Without Contention?

Shared array: *int arr[20];*



With or Without Contention?

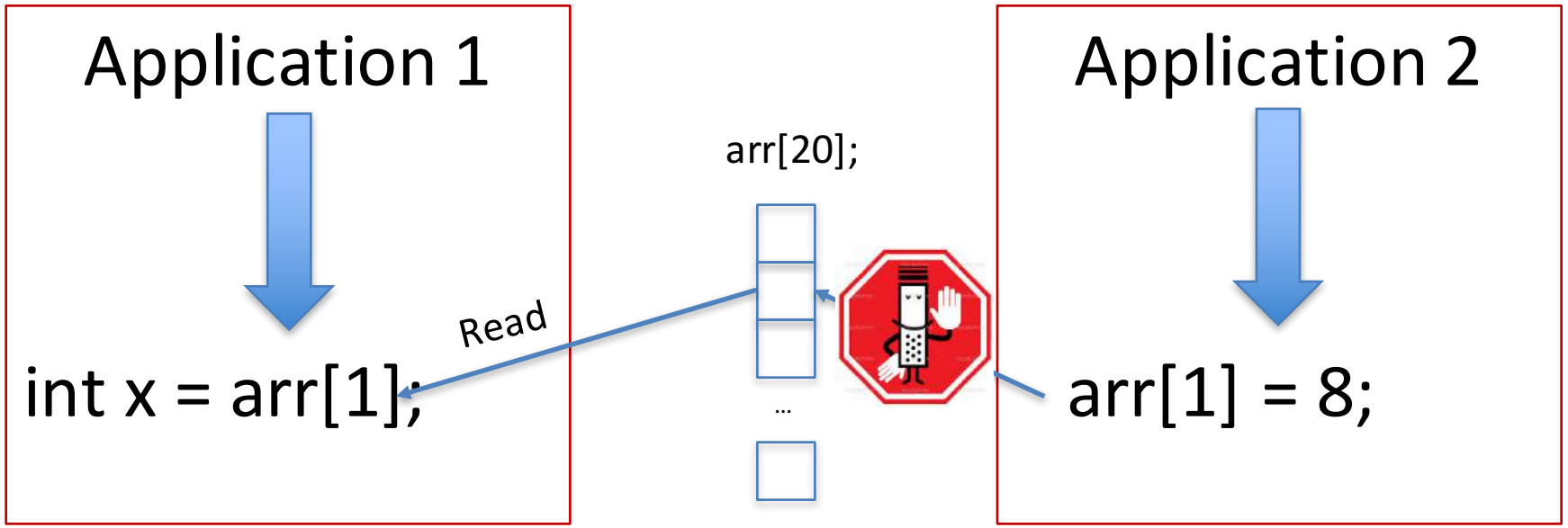
Shared array *int arr[20];*



- Either the Read or Write operation has to stop in order to preserve data coherency
  - Application execution time reduces due to the STOP

With or Without Contention?

Shared array *int arr[20];*



- Either the Read or Write operation has to stop in order to preserve data coherency
  - Application execution time reduces due to the STOP

Performance improvement?

- Example with conflicts:
  - Application execution time is 10s when executes on one CPU (Sequentially)
  - If it executes in parallel:
    - 7s on two CPUs
    - 5.5s on four CPUs
    - 4s on 8 CPUs
    - ….

Parallelism Vs Concurrency

- Informal agreement on the definitions
- Parallel code:
  - executes in parallel and independently and provides performance similar to the ideal
- Concurrent code:
  - Executes in parallel but accessing common shared memory locations, thus performance cannot be ideal

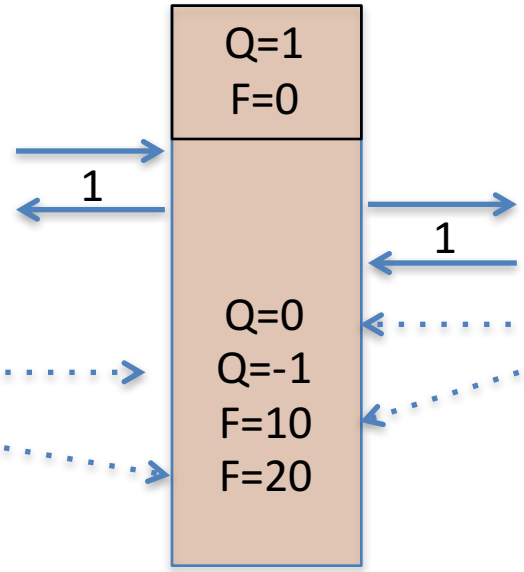
Manage concurrency: real example

BOB

Quantity (Q)=1
Store Fund (F)=0

ALICE

Buy a book {
1) How many available copies of the book are there?
2) Quantity = Quantity - 1
3) Fund = Fund + 10$
}



Buy a book {
1) How many available copies of the book are there?
2) Quantity = Quantity – 1
3) Fund = Fund + 10$
}

Manage concurrency: real example

BOB

Quantity (Q)=1
Store Fund (F)=0

ALICE

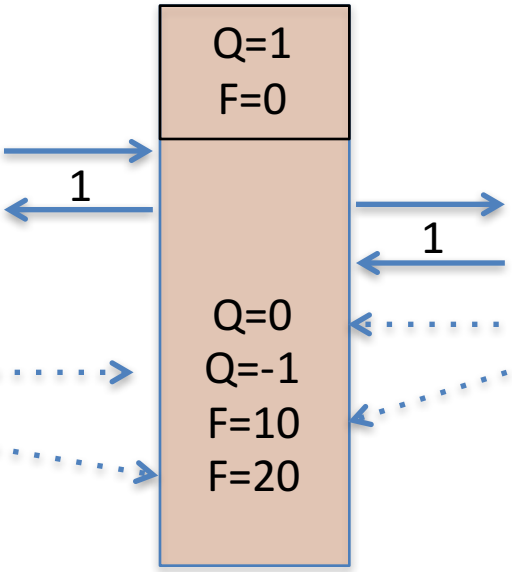
Buy a book {

1) How many available copies of the book are there?

2) Quantity = Quantity - 1

3) Fund = Fund + 10$

}



Buy a b

1) How

the

2) Qua

3) Fun

}

s of

Manage concurrency: real example

BOB

ALICE

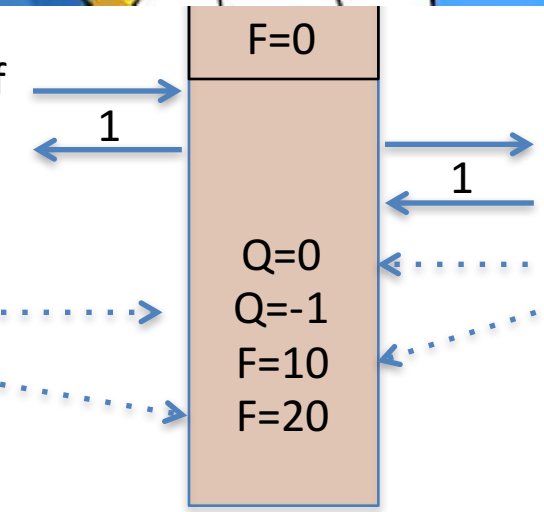
Buy a book {

1) How many available copies of the book are there?

2) Quantity = Quantity - 1

3) Fund = Fund + 10$

}



Buy a b

1) How

the

2) Qua

3) Fun

}

s of

Manage concurrency: real example

BOB

ALICE

Buy a book {

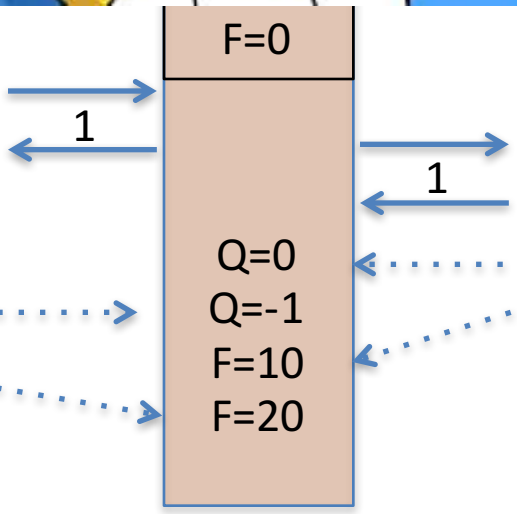
1

2

3

}

ies of



Buy a b

1) How

the

2) Qua

3) Fun

}

s of

Manage concurrency: real example

BOB

Flight = 1 seat available
Hotel = 100 rooms available
Colosseum = 100 tickets

ALICE

Buy a vacation to visit Colosseum

1) Is there any free seat on the flight to Rome (Italy)? Yes!
2) Is there any available room? Yes!
3) Is there any available ticket for visiting the Colosseum? Yes!
4) Reserve Colosseum. Done!
5) Reserve room. Done!
6) Reserve flight. KO!

Buy a vacation to visit Rome

1) Is there any free seat on the flight to Rome (Italy)? Yes!
2) Reserve flight. Done!

Manage concurrency: real example

BOB

Flight = 1 seat available
Hotel = 100 rooms available
Colosseum = 100 tickets

ALICE

Buy a vacation to visit Colosseum

1) Is there any free seat on the flight to Rome (Italy)? Yes!
2) Is there any available room? Yes!
3) Is there any available ticket for visiting the Colosseum? Yes!
4) Reserve Colosseum. Done!
5) Reserve room. Done!
6) Reserve flight. KO!

Buy a va
1) Is th ne
fligh s!
2) Reserve flight. Done!

Manage concurrency: real example

BOB

Flight = 1 seat available
Hotel = 100 rooms available
Colosseum = 100 tickets

ALICE

Buy a vacation to visit Colosseum

1) the
s!
2) m? Yes!
3) et for
Yes!
4) e!
5) Reserve room. Done!
6) Reserve flight. KO!

Buy a va
1) Is th ne
fligh s!
2) Reserve flight. Done!

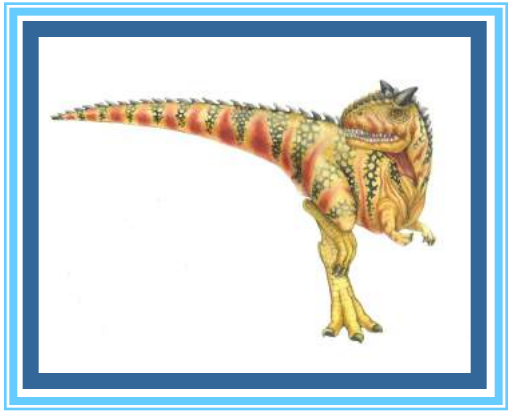
Race condition

- A race condition is an undesirable situation that occurs when a device or system attempts to perform two or more operations at the same time, but because of the nature of the device or system, the operations must be done in the proper sequence to be done correctly.

Lessons

- Parallelism does not entail concurrency
- Manage concurrency is important.
  - If ignored, application's behavior is unpredictable, which means NOT correct
  - If done superficially, it will be easy but performance will be bad!!
  - If done well, it will be painful but performance are great!!

Chapter 3: Processes



Chapter 3: Processes

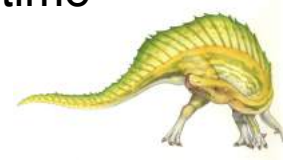
- Process Concept
- Process Scheduling
- Operations on Processes
- Interprocess Communication
- Examples of IPC Systems
- Communication in Client-Server Systems

Objectives

- To introduce the notion of a process -- a program in execution, which forms the basis of all computation
- To describe the various features of processes, including scheduling, creation and termination, and communication
- To explore interprocess communication using shared memory and message passing
- To describe communication in client-server systems

Process Concept

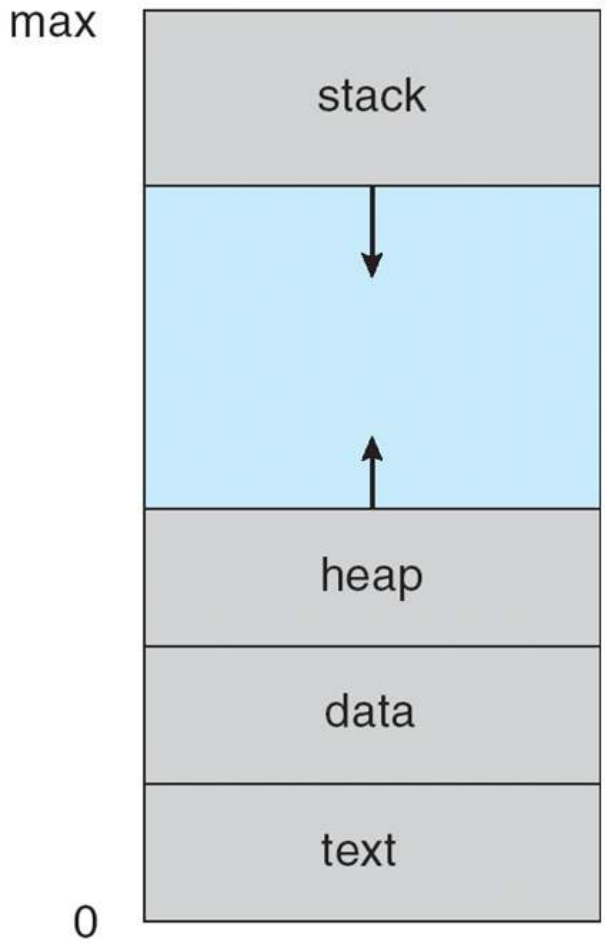
- An operating system executes a variety of programs:
  - Batch system – **jobs**
  - Time-shared systems – **user programs** or **tasks**
- Textbook uses the terms ***job*** and ***process*** almost interchangeably
- **Process** – a program in execution; process execution must progress in sequential fashion
- Multiple parts
  - The program code, also called **text section**
  - Current activity including **program counter**, processor registers
  - **Stack** containing temporary data
    - Function parameters, return addresses, local variables
  - **Data section** containing global variables
  - **Heap** containing memory dynamically allocated during run time



Process Concept (Cont.)

- Program is ***passive*** entity stored on disk (**executable file**), process is ***active***
  - Program becomes process when executable file loaded into memory
- Execution of program started via GUI mouse clicks, command line entry of its name, etc
- One program can be several processes
  - Consider multiple users executing the same program

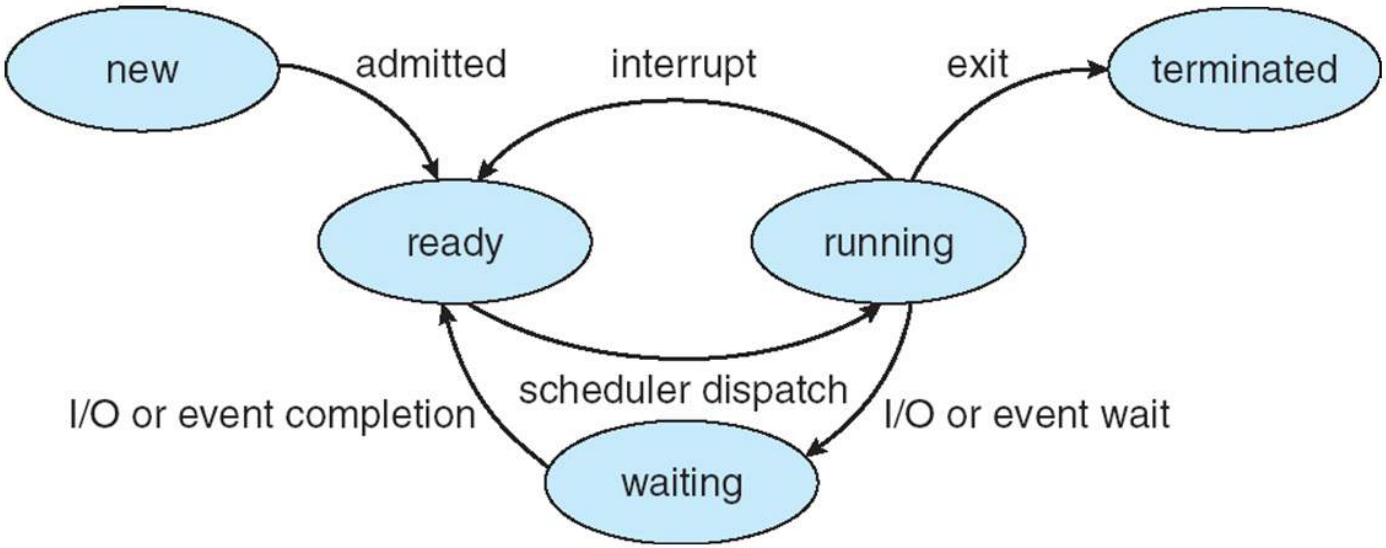
Process in Memory



Process State

- As a process executes, it changes **state**
  - **new**: The process is being created
  - **running**: Instructions are being executed
  - **waiting**: The process is waiting for some event to occur
  - **ready**: The process is waiting to be assigned to a processor
  - **terminated**: The process has finished execution

Diagram of Process State

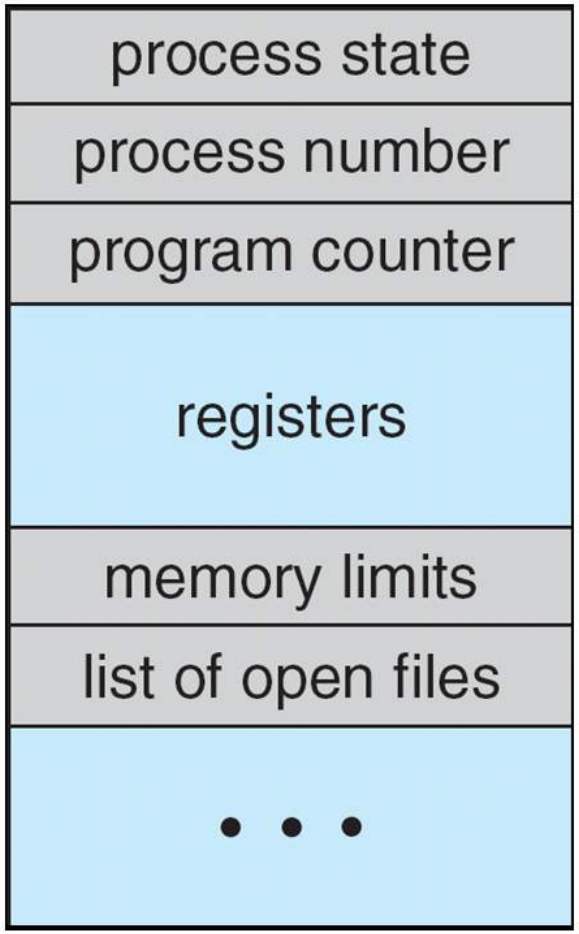


Process Control Block (PCB)

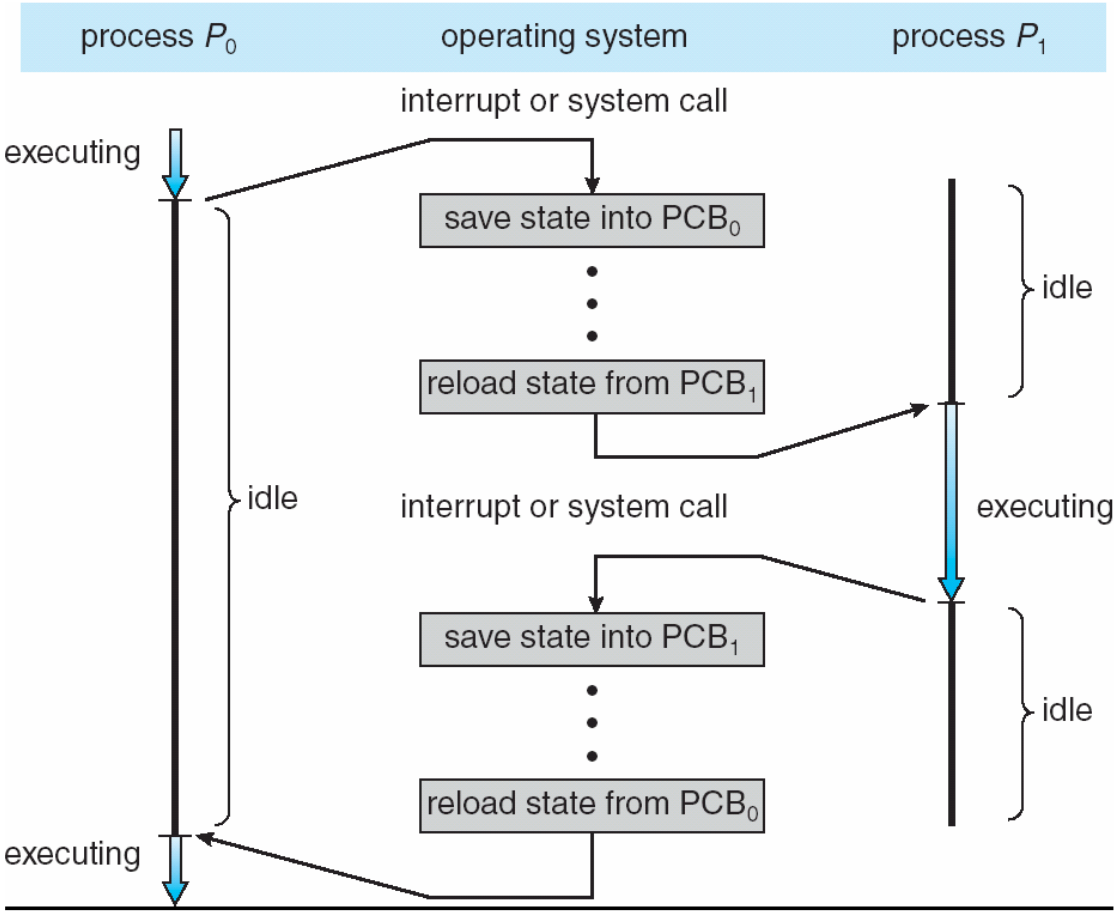
Information associated with each process

(also called **task control block**)

- Process state – running, waiting, etc
- Program counter – location of instruction to next execute
- CPU registers – contents of all process-centric registers
- CPU scheduling information- priorities, scheduling queue pointers
- Memory-management information – memory allocated to the process
- Accounting information – CPU used, clock time elapsed since start, time limits
- I/O status information – I/O devices allocated to process, list of open files



CPU Switch From Process to Process



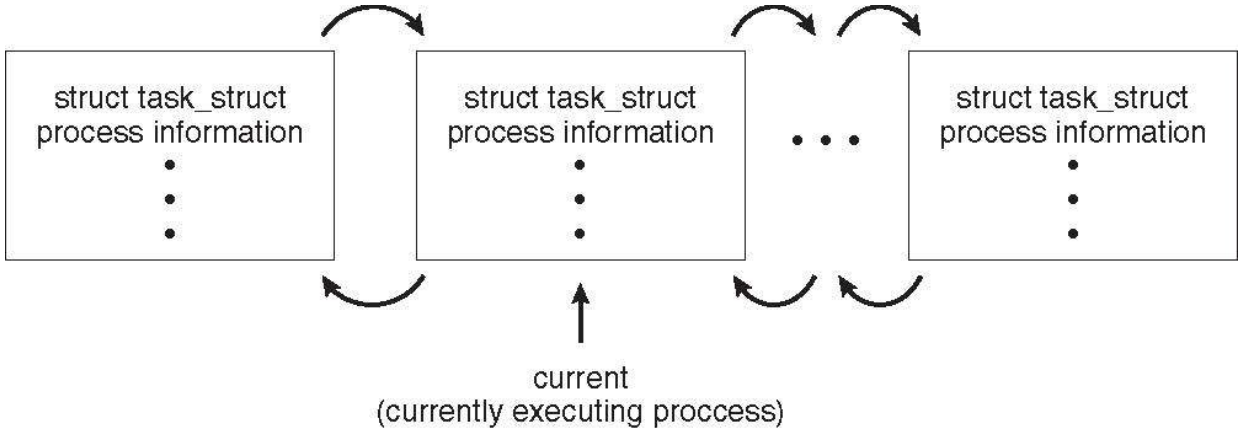
Threads

- So far, process has a single thread of execution
- Consider having multiple program counters per process
  - Multiple locations can execute at once
    - Multiple threads of control -> **threads**
- Must then have storage for thread details, multiple program counters in PCB
- See next chapter

Process Representation in Linux

Represented by the C structure `task_struct`

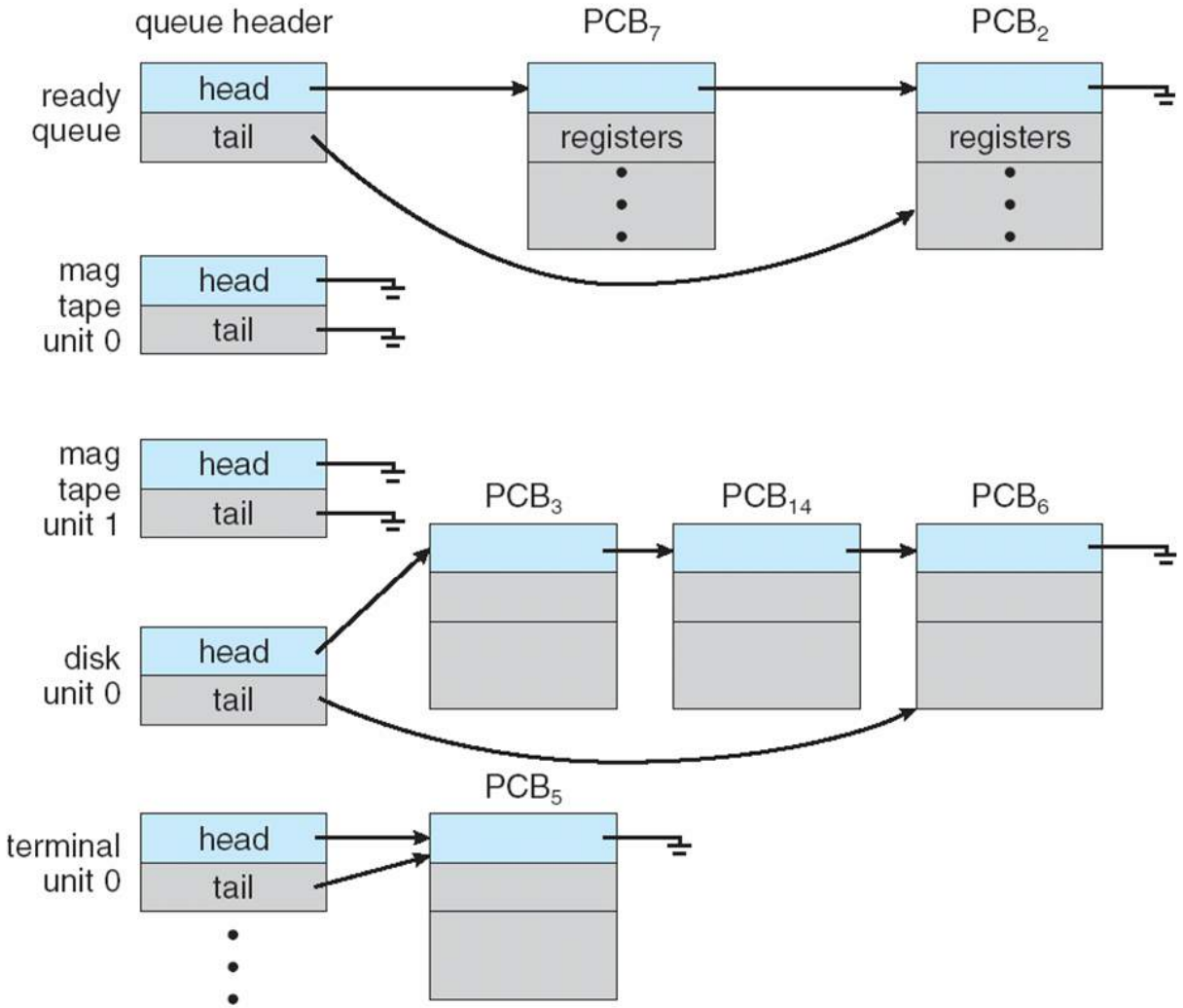
```
pid t_pid; /* process identifier */
long state; /* state of the process */
unsigned int time_slice /* scheduling information */
struct task_struct *parent; /* this process's parent */
struct list_head children; /* this process's children */
struct files_struct *files; /* list of open files */
struct mm_struct *mm; /* address space of this process */
```



Process Scheduling

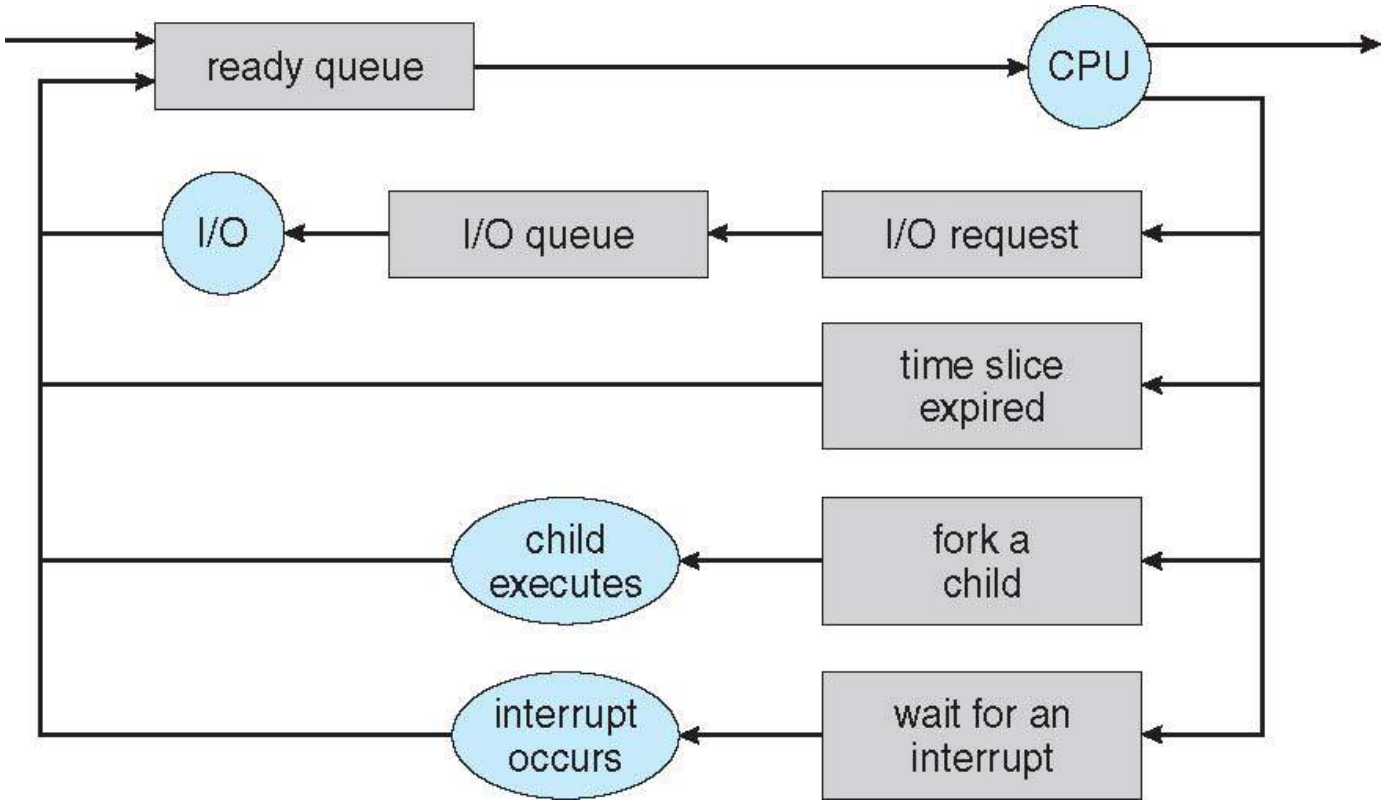
- Maximize CPU use, quickly switch processes onto CPU for time sharing
- **Process scheduler** selects among available processes for next execution on CPU
- Maintains **scheduling queues** of processes
  - **Job queue** – set of all processes in the system
  - **Ready queue** – set of all processes residing in main memory, ready and waiting to execute
  - **Device queues** – set of processes waiting for an I/O device
  - Processes migrate among the various queues

Ready Queue And Various I/O Device Queues



Representation of Process Scheduling

- **Queueing diagram** represents queues, resources, flows

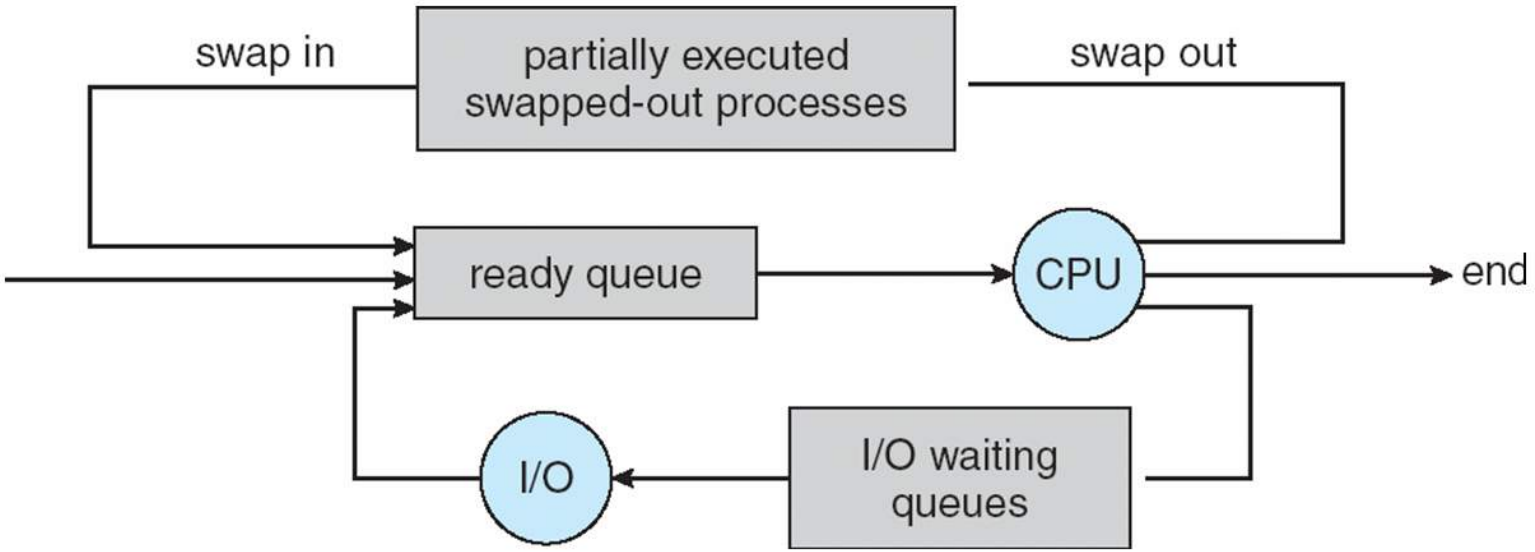


Schedulers

- **Short-term scheduler** (or **CPU scheduler**) – selects which process should be executed next and allocates CPU
  - Sometimes the only scheduler in a system
  - Short-term scheduler is invoked frequently (milliseconds) ⇒ (must be fast)
- **Long-term scheduler** (or **job scheduler**) – selects which processes should be brought into the ready queue
  - Long-term scheduler is invoked infrequently (seconds, minutes) ⇒ (may be slow)
  - The long-term scheduler controls the **degree of multiprogramming**
- Processes can be described as either:
  - **I/O-bound process** – spends more time doing I/O than computations, many short CPU bursts
  - **CPU-bound process** – spends more time doing computations; few very long CPU bursts
- Long-term scheduler strives for good ***process mix***

Addition of Medium Term Scheduling

- **Medium-term scheduler** can be added if degree of multiple programming needs to decrease
  - Remove process from memory, store on disk, bring back in from disk to continue execution: **swapping**



Context Switch

- When CPU switches to another process, the system must **save the state** of the old process and load the **saved state** for the new process via a **context switch**
- **Context** of a process represented in the PCB
- Context-switch time is overhead; the system does no useful work while switching
  - The more complex the OS and the PCB ➔ the longer the context switch
- Time dependent on hardware support
  - Some hardware provides multiple sets of registers per CPU ➔ multiple contexts loaded at once

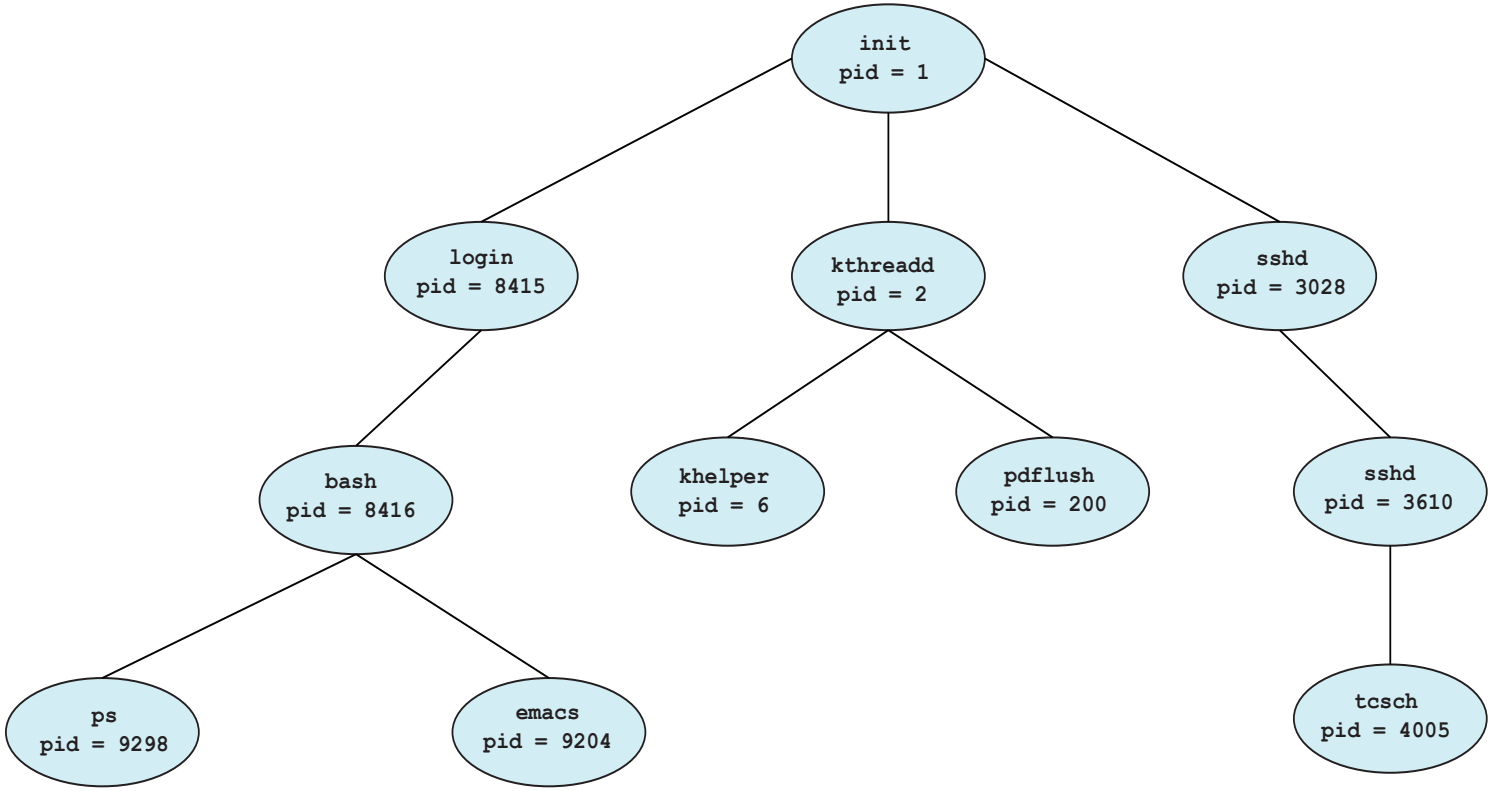
Operations on Processes

- System must provide mechanisms for:
  - process creation,
  - process termination,
  - and so on as detailed next

Process Creation

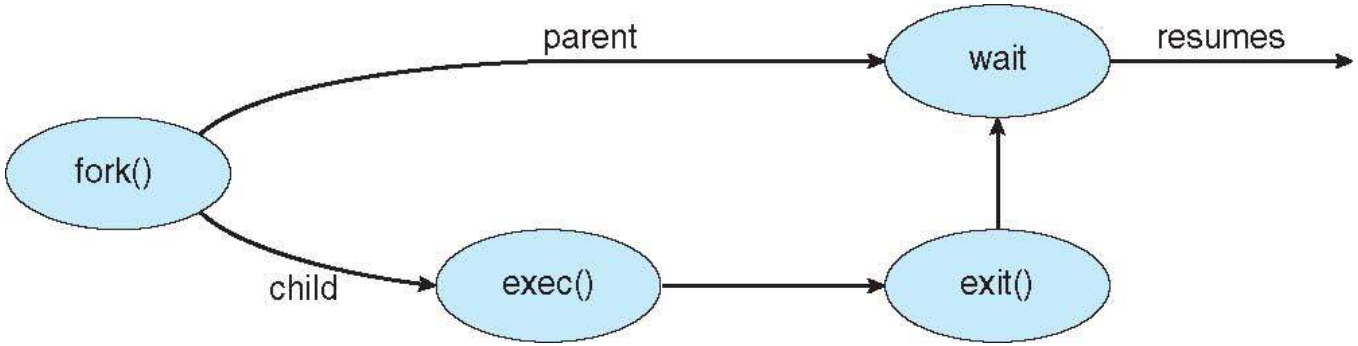
- **Parent** process create **children** processes, which, in turn create other processes, forming a **tree** of processes
- Generally, process identified and managed via a **process identifier** (**pid**)
- Resource sharing options
  - Parent and children share all resources
  - Children share subset of parent's resources
  - Parent and child share no resources
- Execution options
  - Parent and children execute concurrently
  - Parent waits until children terminate

A Tree of Processes in Linux



Process Creation (Cont.)

- Address space
  - Child duplicate of parent
  - Child has a program loaded into it
- UNIX examples
  - `fork()` system call creates new process
  - `exec()` system call used after a `fork()` to replace the process' memory space with a new program



C Program Forking Separate Process

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

int main()
{
pid_t pid;

    /* fork a child process */
    pid = fork();

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

    return 0;
}
```

Process Termination

- Process executes last statement and then asks the operating system to delete it using the **`exit()`** system call.
  - Returns status data from child to parent (via **`wait()`**)
  - Process' resources are deallocated by operating system
- Parent may terminate the execution of children processes using the **`abort()`** system call. Some reasons for doing so:
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - The parent is exiting and the operating systems does not allow a child to continue if its parent terminates

Process Termination

- Some operating systems do not allow child to exists if its parent has terminated. If a process terminates, then all its children must also be terminated.
  - **cascading termination.** All children, grandchildren, etc. are terminated.
  - The termination is initiated by the operating system.
- The parent process may wait for termination of a child process by using the `wait()` system call. The call returns status information and the pid of the terminated process

```
pid = wait(&status);
```

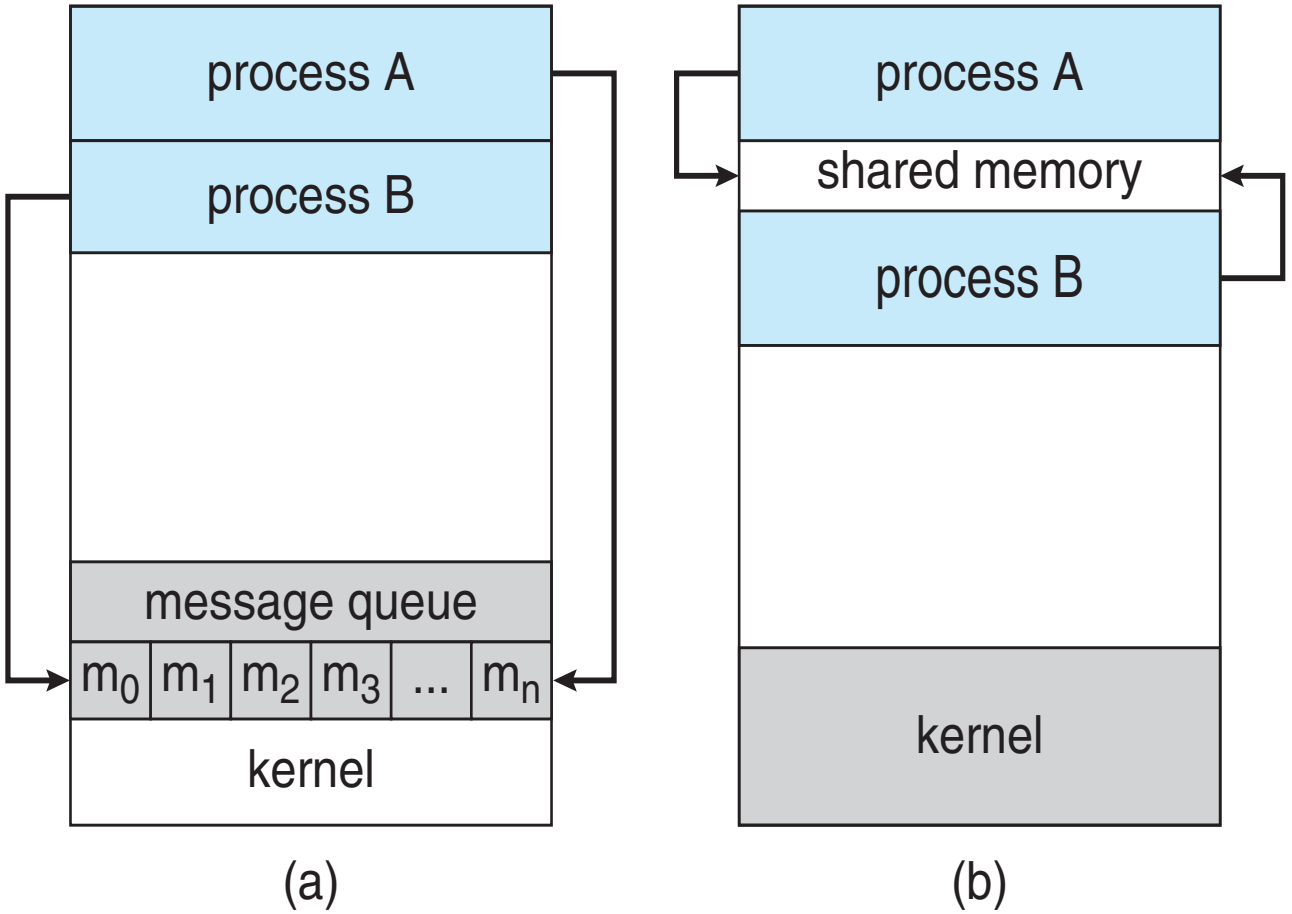
- If no parent waiting (did not invoke `wait()`) process is a **zombie**
- If parent terminated without invoking `wait`, process is an **orphan**

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**

Communications Models

(a) Message passing. (b) shared memory.



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

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.
- Synchronization is discussed in great details in Chapter 5.

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
  - **bounded-buffer** assumes that there is a fixed buffer size

Bounded-Buffer – Shared-Memory Solution

- Shared data

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

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

- 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 – 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

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

Indirect Communication

- 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

- 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.

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**

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 */
}
```

Buffering

- Queue of messages attached to the link.
- 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

Examples of IPC Systems - POSIX

- POSIX Shared Memory
  - Process first creates shared memory segment
    ```
    shm_fd = shm_open(name, O CREAT | O RDWR, 0666);
    ```
  - Also used to open an existing segment to share it
  - Set the size of the object
    ```
    ftruncate(shm fd, 4096);
    ```
  - Now the process could write to the shared memory
    ```
    sprintf(shared memory, "Writing to shared
    memory");
    ```

IPC POSIX Producer

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main()
{
/* the size (in bytes) of shared memory object */
const int SIZE = 4096;
/* name of the shared memory object */
const char *name = "OS";
/* strings written to shared memory */
const char *message_0 = "Hello";
const char *message_1 = "World!";

/* shared memory file descriptor */
int shm_fd;
/* pointer to shared memory obect */
void *ptr;

    /* create the shared memory object */
    shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);

    /* configure the size of the shared memory object */
    ftruncate(shm_fd, SIZE);

    /* memory map the shared memory object */
    ptr = mmap(0, SIZE, PROT_WRITE, MAP_SHARED, shm_fd, 0);

    /* write to the shared memory object */
    sprintf(ptr,"%s",message_0);
    ptr += strlen(message_0);
    sprintf(ptr,"%s",message_1);
    ptr += strlen(message_1);

    return 0;
}
```

IPC POSIX Consumer

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

int main()
{
/* the size (in bytes) of shared memory object */
const int SIZE = 4096;
/* name of the shared memory object */
const char *name = "OS";
/* shared memory file descriptor */
int shm_fd;
/* pointer to shared memory obect */
void *ptr;

    /* open the shared memory object */
    shm_fd = shm_open(name, O_RDONLY, 0666);

    /* memory map the shared memory object */
    ptr = mmap(0, SIZE, PROT_READ, MAP_SHARED, shm_fd, 0);

    /* read from the shared memory object */
    printf("%s",(char *)ptr);

    /* remove the shared memory object */
    shm_unlink(name);

    return 0;
}
```

Possible Design for Homework 4

User provides Process ID

Create the receiving Mailbox:
mq_open(ID, O_NONBLOCK | O_RDONLY | O_CREAT | O_EXCL, 0666, &attr);
Example: Process P0 has ID = /70

```
struct mq_attr attr;
    attr.mq_flags = O_NONBLOCK;
    attr.mq_maxmsg = 10;
    attr.mq_msgsize = 15;
    attr.mq_curmsgs = 0;
```

Build mapping between communicating processes
Example: P0 informs P1 and P2 about its Mailbox ID

Pass mapping to a component that handles the Communications and finishes once the algorithm is over

Process P0 starts!

Heads-up working with Mailboxes

- When the program is over, don't forget to:

  mq_close(mqMailbox);

  mq_unlink(ID);

- POSIX message queues have kernel persistence: if not removed by mq_unlink(…), a message queue will exist until the system is shut down.
- If you do not remove it, or your program terminates unexpectedly, the created mailboxes are still there. Make sure to handle this case once you open it.
  - Always check the return value of the mq_open system call

Possible Design for Homework 4

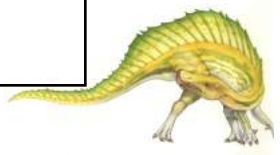
Mailbox Manager

Am I P0? — Yes →

↓ No

Wait for a message to be received on the process's Mailbox(the Read-only one):
mq_receive(m_Mailbox, &msg_buf[0], 100, 0);
The call may return immediately if O_NONBLOCK has been specified, thus wrap the call into a loop

```
while(1) {
        if(m_Exit) {
            break;
        }
        msg_length = mq_receive(m_Mailbox, &msg_buf[0], 100, 0);
        if(msg_length >= 0) {
            processMsg(msg_buf);
        } else if(errno != EAGAIN){
            cout << "Error: " << strerror(errno) << endl;
            exit(-1);
        }
    }
```



Once a temperature is received, you compute some simple math function and propagate the result to other processes (write to other processes' Mailboxes).

If it's the first time, then you first open the needed mailboxes for writing
mailbox = mq_open(id, O_WRONLY);

Example: P1 writes to P0's, P3's, and P4's Mailboxes
You can use the mapping built at the beginning to know the Mailboxes each process is interested to

To send the result of the math function:
mq_send(dest, msg_buf, length+1, 0);

Possible Design for Homework 4

Mailbox Manager

Am I P0?

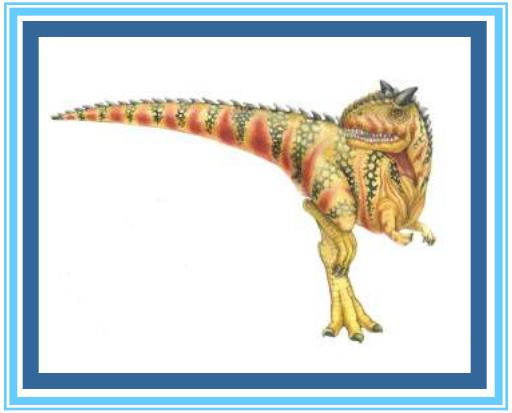
Yes

Open writing Mailboxes:
mq_open(id, O_WRONLY);
Here you do not create a mailbox, you open one in write mode. If the open returns error, then it was not there and you need to wait/retry.

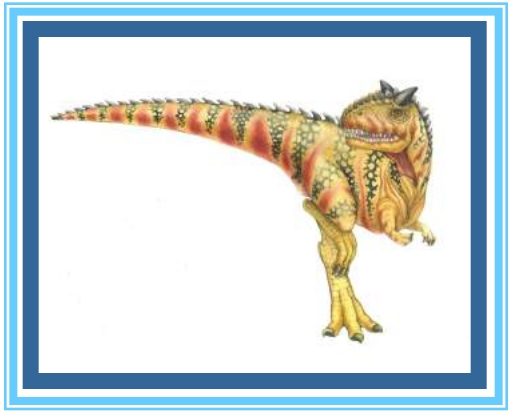
Send temperature to P1 and P2

Wait for messages from P1 and P2.
Basically P0 starts behaving as all other processes

End of Chapter 3



Chapter 4: Threads



Chapter 4: Threads

- Overview
- Multicore Programming
- Multithreading Models
- Thread Libraries
- Implicit Threading
- Threading Issues
- Operating System Examples

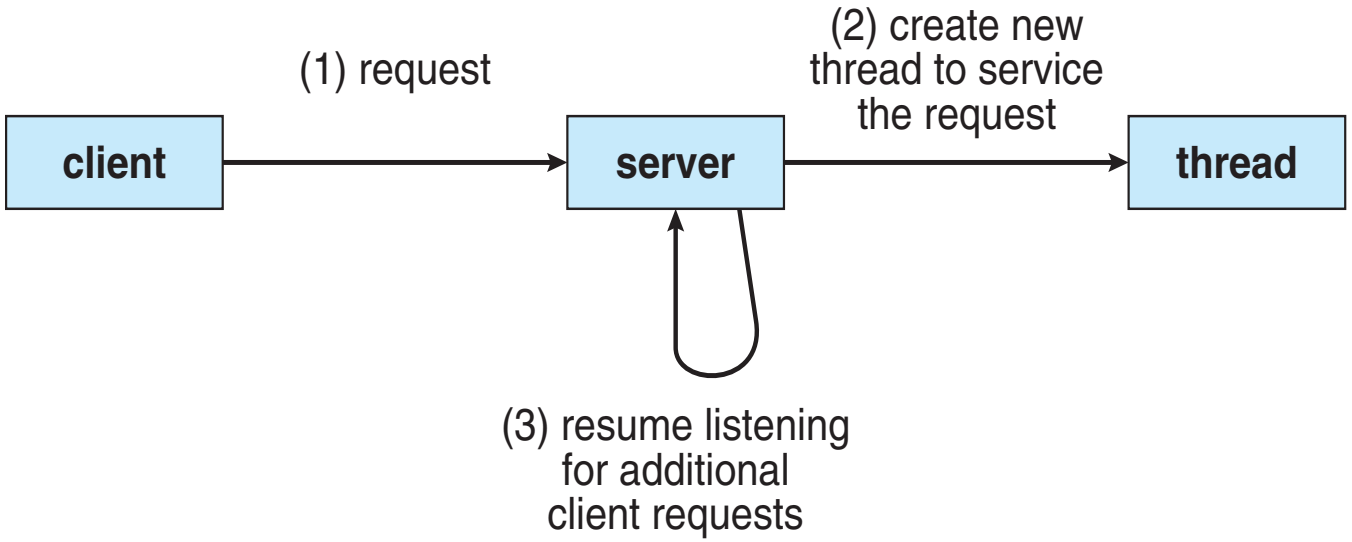
Objectives

- To introduce the notion of a thread—a fundamental unit of CPU utilization that forms the basis of multithreaded computer systems
- To discuss the APIs for the Pthreads and Windows libraries
- To explore several strategies that provide implicit threading
- To examine issues related to multithreaded programming
- To cover operating system support for threads in Windows and Linux

Motivation

- Most modern applications are multithreaded
- Threads run within application
- Multiple tasks with the application can be implemented by separate threads
  - Update display
  - Fetch data
  - Spell checking
  - Answer a network request
- Process creation is heavy-weight while thread creation is light-weight
- Can simplify code, increase efficiency
- Kernels are generally multithreaded

Multithreaded Server Architecture



Benefits

- **Responsiveness –** may allow continued execution if part of process is blocked, especially important for user interfaces
- **Resource Sharing –** threads share resources of process, easier than shared memory or message passing
- **Economy –** cheaper than process creation, thread switching lower overhead than context switching
- **Scalability –** process can take advantage of multiprocessor architectures

Multicore Programming

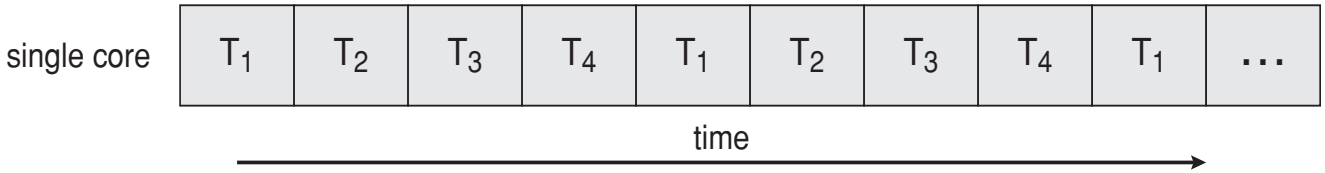
- **Multicore** or **multiprocessor** systems putting pressure on programmers, challenges include:
  - **Dividing activities**
  - **Balance**
  - **Data splitting**
  - **Data dependency**
  - **Testing and debugging**
- ***Parallelism*** implies a system can perform more than one task simultaneously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

Multicore Programming (Cont.)

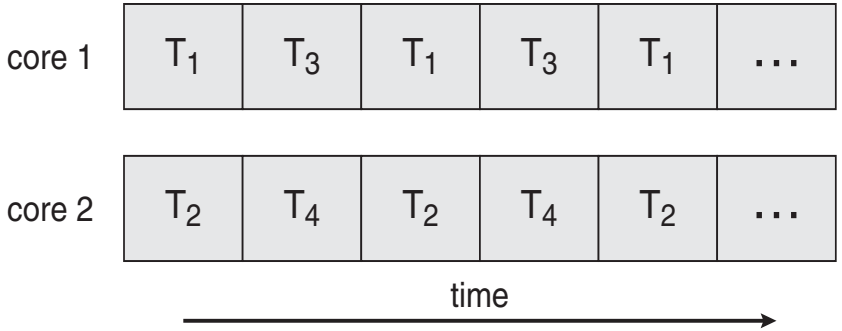
- Types of parallelism
  - **Data parallelism** – distributes subsets of the same data across multiple cores, same operation on each
  - **Task parallelism** – distributing threads across cores, each thread performing unique operation
- As # of threads grows, so does architectural support for threading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

Concurrency vs. Parallelism

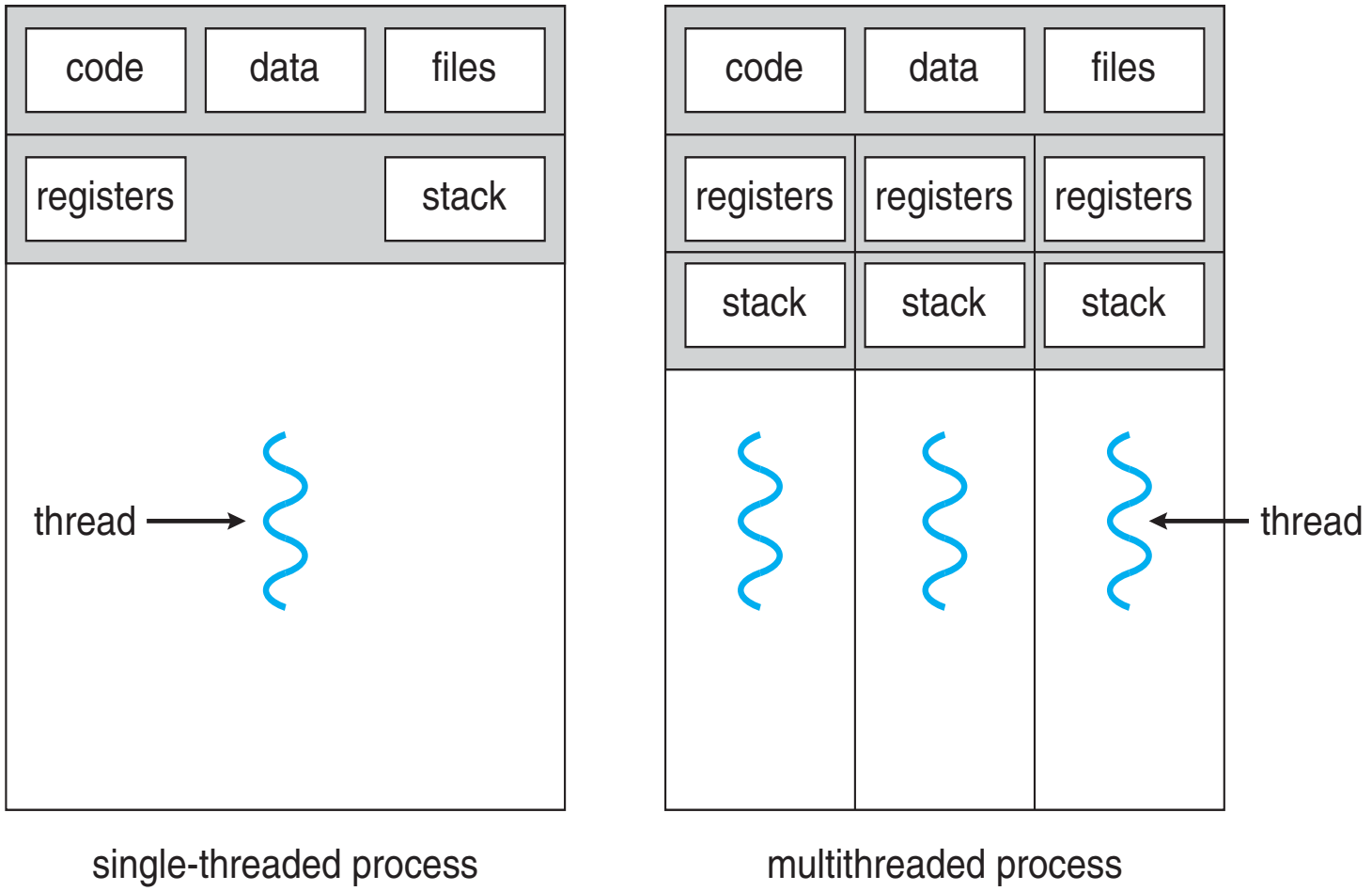
- **Concurrent execution on single-core system:**



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



Single and Multithreaded Processes



single-threaded process

multithreaded process

Amdahl's Law

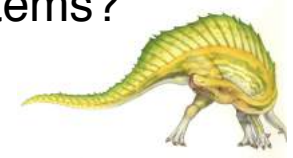
- Identifies performance gains from adding additional cores to an application that has both serial and parallel components
- *S* is serial portion
- *N* processing cores

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

- 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?



User Threads and Kernel Threads

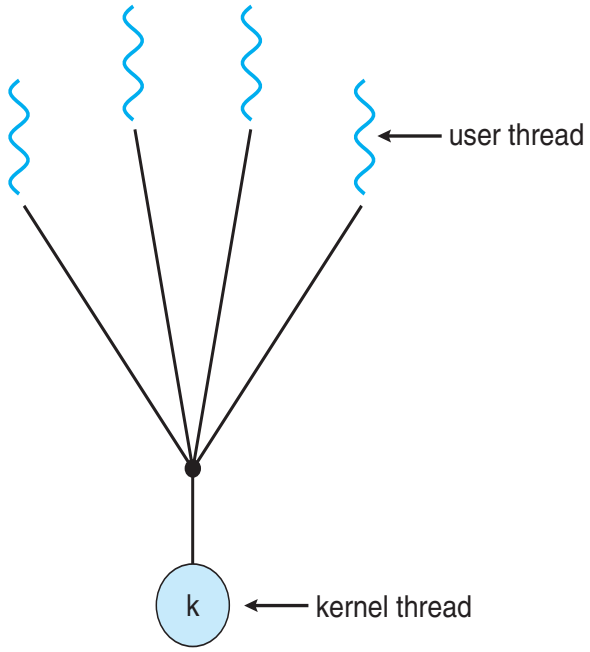
- **User threads** - management done by user-level threads library
- Three primary thread libraries:
  - POSIX **Pthreads**
  - Windows threads
  - Java threads
- **Kernel threads** - Supported by the Kernel
- Examples – virtually all general purpose operating systems, including:
  - Windows
  - Solaris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

Multithreading Models

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

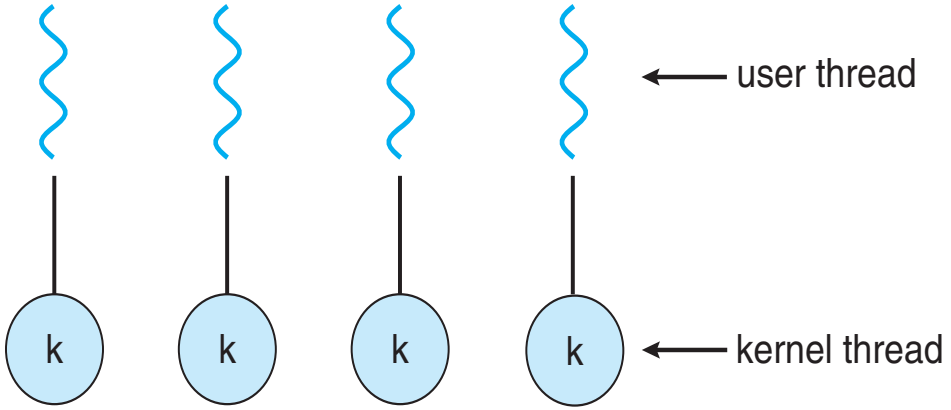
Many-to-One

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



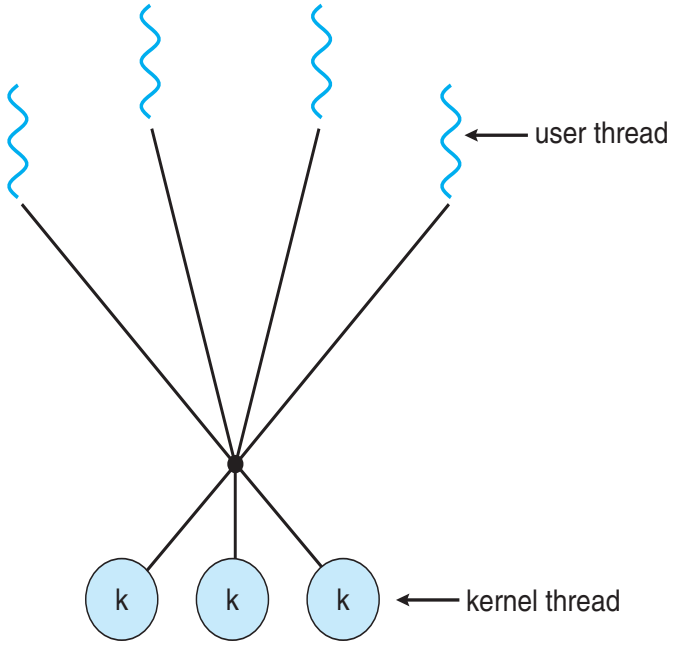
One-to-One

- 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



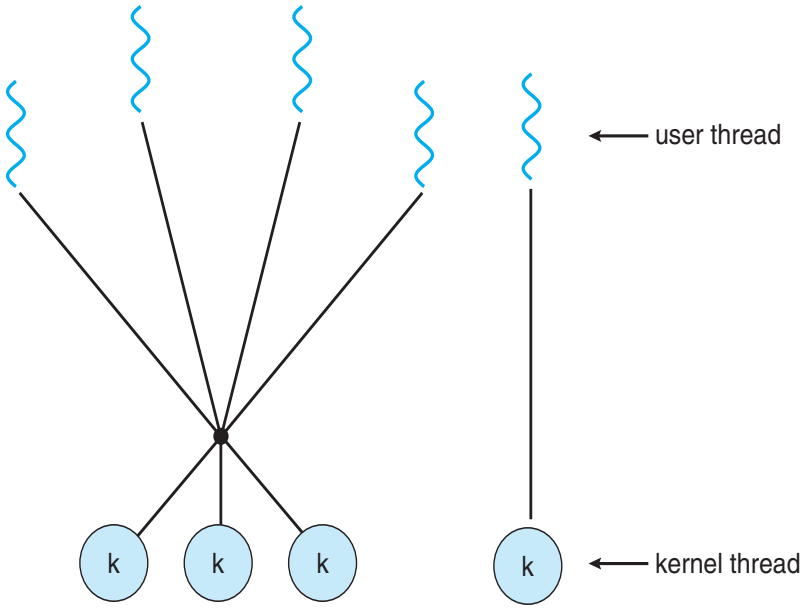
Many-to-Many Model

- 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



Thread Libraries

- **Thread library** provides programmer with API for creating and managing threads
- Two primary ways of implementing
  - Library entirely in user space
  - Kernel-level library supported by the OS

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)

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 (Cont.)

```
   /* 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);
}
```

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 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);
   }
}
```

Implicit Threading

- Growing in popularity as numbers of threads increase, program correctness more difficult with explicit threads
- Creation and management of threads done by compilers and run-time libraries rather than programmers
- Three methods explored
  - Thread Pools
  - OpenMP
  - Grand Central Dispatch
- Other methods include Microsoft Threading Building Blocks (TBB), `java.util.concurrent` package

Thread Pools

- Create a number of threads in a pool where they await work
- Advantages:
  - Usually slightly faster to service a request with an existing thread than create a new thread
  - Allows the number of threads in the application(s) to be bound to the size of the pool
  - Separating task to be performed from mechanics of creating task allows different strategies for running task
    - i.e.Tasks could be scheduled to run periodically

OpenMP

- Set of compiler directives and an API for C, C++, FORTRAN
- Provides support for parallel programming in shared-memory environments
- Identifies **parallel regions** – blocks of code that can run in parallel

```
#pragma omp parallel
```

Create as many threads as there are cores

```
#pragma omp parallel for
   for(i=0;i<N;i++) {
     c[i] = a[i] + b[i];
}
```

Run for loop in parallel

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

int main(int argc, char *argv[])
{
   /* sequential code */

   #pragma omp parallel
   {
      printf("I am a parallel region.");
   }

   /* sequential code */

   return 0;
}
```

Threading Issues

- Semantics of **fork()** and **exec()** system calls
- Signal handling
  - Synchronous and asynchronous
- Thread cancellation of target thread
  - Asynchronous or deferred
- Thread-local storage
- Scheduler Activations

Semantics of fork() and exec()

- Does `fork()` duplicate only the calling thread or all threads?
  - Some UNIXes have two versions of fork
- `exec()` usually works as normal – replace the running process including all threads

Signal Handling

- **Signals** are used in UNIX systems to notify a process that a particular event has occurred.
- A **signal handler** is used to process signals
  1. Signal is generated by particular event
  2. Signal is delivered to a process
  3. Signal is handled by one of two signal handlers:
     1. default
     2. user-defined
- Every signal has **default handler** that kernel runs when handling signal
  - **User-defined signal handler** can override default
  - For single-threaded, signal delivered to process

Signal Handling (Cont.)

- Where should a signal be delivered for multi-threaded?
  - Deliver the signal to the thread to which the signal applies
  - Deliver the signal to every thread in the process
  - Deliver the signal to certain threads in the process
  - Assign a specific thread to receive all signals for the process
- The standard UNIX function for delivering a signal is

  kill(pid t pid, int signal)
- Pthreads provides the following function, which allows a signal to be delivered to a specified thread (tid):

  pthread kill(pthread t tid, int signal)

Thread Cancellation

- Terminating a thread before it has finished
- Thread to be canceled is **target thread**
- Two general approaches:
  - **Asynchronous cancellation** terminates the target thread immediately
  - **Deferred cancellation** allows the target thread to periodically check if it should be cancelled
- Pthread code to create and cancel a thread:

```
pthread_t tid;

/* create the thread */
pthread_create(&tid, 0, worker, NULL);

. . .

/* cancel the thread */
pthread_cancel(tid);
```

Thread Cancellation (Cont.)

- Invoking thread cancellation requests cancellation, but actual cancellation depends on thread state

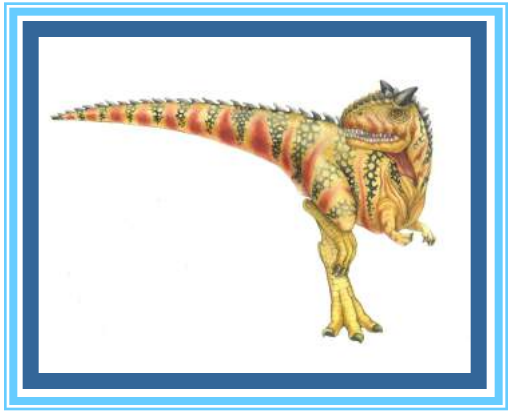
Mode	State	Type
Off	Disabled	–
Deferred	Enabled	Deferred
Asynchronous	Enabled	Asynchronous

- If thread has cancellation disabled, cancellation remains pending until thread enables it
- Default type is deferred
  - Cancellation only occurs when thread reaches **cancellation point**
    - I.e. `pthread_testcancel()`
    - Then **cleanup handler** is invoked
- On Linux systems, thread cancellation is handled through signals

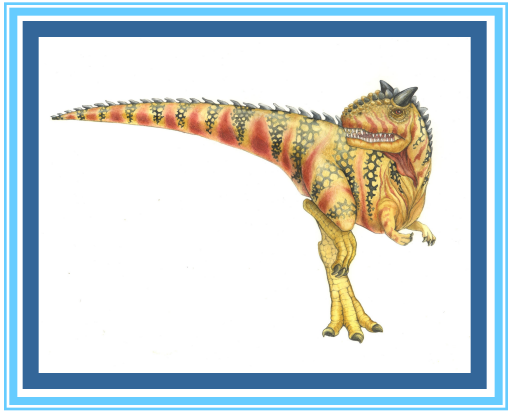
Thread-Local Storage

- **Thread-local storage** (**TLS**) allows each thread to have its own copy of data
- Useful when you do not have control over the thread creation process (i.e., when using a thread pool)
- Different from local variables
  - Local variables visible only during single function invocation
  - TLS visible across function invocations
- Similar to `static` data
  - TLS is unique to each thread

End of Chapter 4



Chapter 5: Process Synchronization



Chapter 5: Process Synchronization

- Background
- The Critical-Section Problem
- Peterson's Solution
- Synchronization Hardware
- Mutex Locks
- Semaphores
- Classic Problems of Synchronization
- Monitors
- Synchronization Examples
- Alternative Approaches

Objectives

- To present the concept of process synchronization.
- To introduce the critical-section problem, whose solutions can be used to ensure the consistency of shared data
- To present both software and hardware solutions of the critical-section problem
- To examine several classical process-synchronization problems
- To explore several tools that are used to solve process synchronization problems

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;
        counter++;
}
```

Consumer

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

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}

Intuition of why it is wrong

- High-level operations have to appear as executed serially, one after the other
- In the producer-consumer case, it can be either:
  - Consumer before Producer, therefore:
    - `counter++ and then counter--`
  - Producer before Consumer, therefore:
    - `counter-- and then counter++`

Critical Section Problem

- Consider system of ***n*** processes {p_0, p_1, … 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**

Critical Section

- General structure of process P_i

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

Idea Algorithm for Process P_i

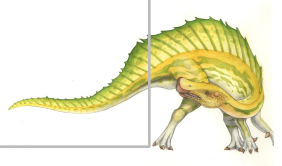
```
do {

    while (turn == j);

            critical section

    turn = j;

            remainder section

} while (true);
```

Why simple solutions DO NOT work

CS_status = True (CS empty) or False (CS busy)

```
        Process Pi
if(CS_status == False)
   CS_status == True
        critical section
   CS_status == False
        remainder section
```

```
                Execution:
P0: if(CS_status == False) [CS_status is False]
P1: if(CS_status == False) [CS_status is False]
   P0: CS_status == True [CS_status is True]
   P1: CS_status == True [CS_status is True]
          P0: Enters critical section
          P1: Enters critical section
```



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
- **Non-preemptive** – runs until exits kernel mode, blocks, or voluntarily yields CPU
  - Essentially free of race conditions in kernel mode

Peterson's Solution

- Good algorithmic description of solving the problem
- Two process solution
- Assume that the **`load`** and **`store`** machine-language instructions are atomic; that is, cannot be interrupted
- The two processes share two variables:
  - **`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

```
do {
    flag[i] = true;
    turn = j;
    while (flag[j] && turn = = j);

            critical section

    flag[i] = false;

            remainder section

 } while (true);
```

Algorithm in action

```
Init:
flag[0] = flag[1] = false;
```

P0

```
1: flag[0] = true;
2: turn = 1;
3: while (flag[1] && turn = = 1);
  ...critical section...
4: flag[0] = false;
  ...remainder section...
```

P1

```
1: flag[1] = true;
2: turn = 0;
3: while (flag[0] && turn = = 0);
  ...critical section...
4: flag[1] = false;
  ...remainder section...
```

Peterson's Solution (Cont.)

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

     P_i enters CS only if:

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

Synchronization Hardware

- Many systems provide hardware support for implementing the critical section code.
- All solutions below based on idea of **locking**
  - Protecting critical regions via locks
- 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

Solution to Critical-section Problem Using Locks

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

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));
   /* critical section */
    lock = false;
           /* remainder section */
 } while (true);
```

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;
}
```

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.

Solution using compare_and_swap

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

```
do {
     while (compare_and_swap(&lock, 0, 1) != 0);
   /* critical section */
lock = 0;
   /* remainder section */
} while (true);
```

Bounded-waiting Mutual Exclusion with test_and_set

```
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);
```

Mutex Locks

- Previous 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**

acquire() and release()

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

```
release() {
    available = true;
}
```

```
do {
    acquire lock
        critical section
    release lock
      remainder section
} while (true);
```

Possible Design HW5

Application starts

Create a structure or a class for representing a matrix given the representation on the file.
For example:

```
struct matrix {
        int row;
        int col;
        bool decimal;
        QVector<QVector<double>>* decVector;
};
```

Application creates two threads to read the two input files:

*pthread_create(&readThreads[0], &attr, readFile, (void *)argv[1]);*
*pthread_create(&readThreads[1], &attr, readFile, (void *)argv[2]);*
Example: argv[1] is the input file name for the Matrix A

Possible Design HW5

Wait for threads to finish and validate the values in the matrix. Always check the return values of pthread calls!!!

To fetch the matrices you can use either two global variables (no concurrency) or use the pthread_exit function to pass the just populated matrices (for examle: pthread_exit((void*) matrixObj);

Example:

```
int rc;
rc = pthread_join(readThreads[0], (void**)&matrixA);
if (rc) {
        printf("ERROR return code from pthread_join() is %s\n",
        strerror(rc));
        exit(-1);
}
```

Possible Design HW5

Activate as many threads as the size of the final matrix

EXAMPLE:

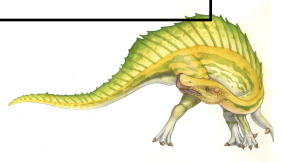
```
int status;
for(int i=0;i<numThreads;i++) {
        rc = pthread_join(mThreads[i], (void **)&status);
        if (rc) {
                printf("ERROR return code from pthread_join() is %s\n", strerror(rc));
                exit(-1);
        }
}
```

Each thread computes $[C_{i,j}]$ and writes to the "temp.txt" file: $C_{i,j}$=V on a new line

EXAMPLE

$C_{1,2}$=16.

FILE cannot be written concurrently! Mutual exclusion should be implemented to prevent more than one thread to write on the file.



Possible Design HW5

temp.txt will be something like:

$C_{1,2}$=16
$C_{3,1}$=54
$C_{4,1}$=87
$C_{1,3}$=6
…

The main thread joins the execution of all the processing threads and re-format the temp.txt file so that the output file will look like one of the input files. In other words, each line of the file describes each row of the output matrix.

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 Usage

- **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;
```

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);
    }
}
```

Deadlock and Starvation

- **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	P_1
`wait(S);`	`wait(Q);`
`wait(Q);`	`wait(S);`
`...`	`...`
`signal(S);`	`signal(Q);`
`signal(Q);`	`signal(S);`

- **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**

Classical Problems of Synchronization

- Classical problems used to test newly-proposed synchronization schemes
  - Bounded-Buffer Problem
  - Readers and Writers Problem
  - Dining-Philosophers Problem

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(mutex);
        ...
        /* add next produced to the buffer */
        ...
    signal(mutex);
    signal(full);
} while (true);
```

Bounded Buffer Problem (Cont.)

- The structure of the consumer process

```
Do {
    wait(full);
    wait(mutex);
       ...
    /* remove an item from buffer to next_consumed */
       ...
    signal(mutex);
    signal(empty);
       ...
    /* consume the item in next consumed */
       ...
} while (true);
```

Readers-Writers Problem

- A data set is shared among a number of concurrent processes
  - Readers – only read the data set; they do ***not*** perform any updates
  - Writers – can both read and write
- Problem – allow multiple readers to read at the same time
  - Only one single writer can access the shared data at the same time
- 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 **`mutex`** initialized to 1
  - 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.)

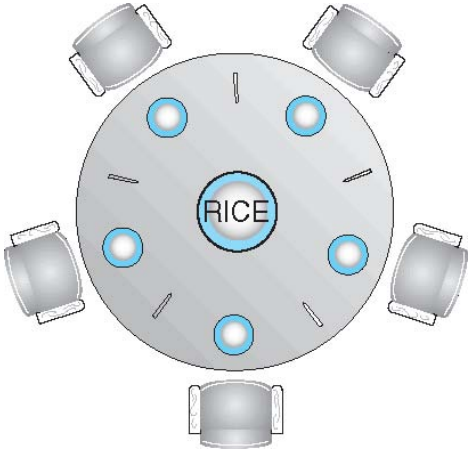
- The structure of a reader process

```
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 forks 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.

Problems with Semaphores

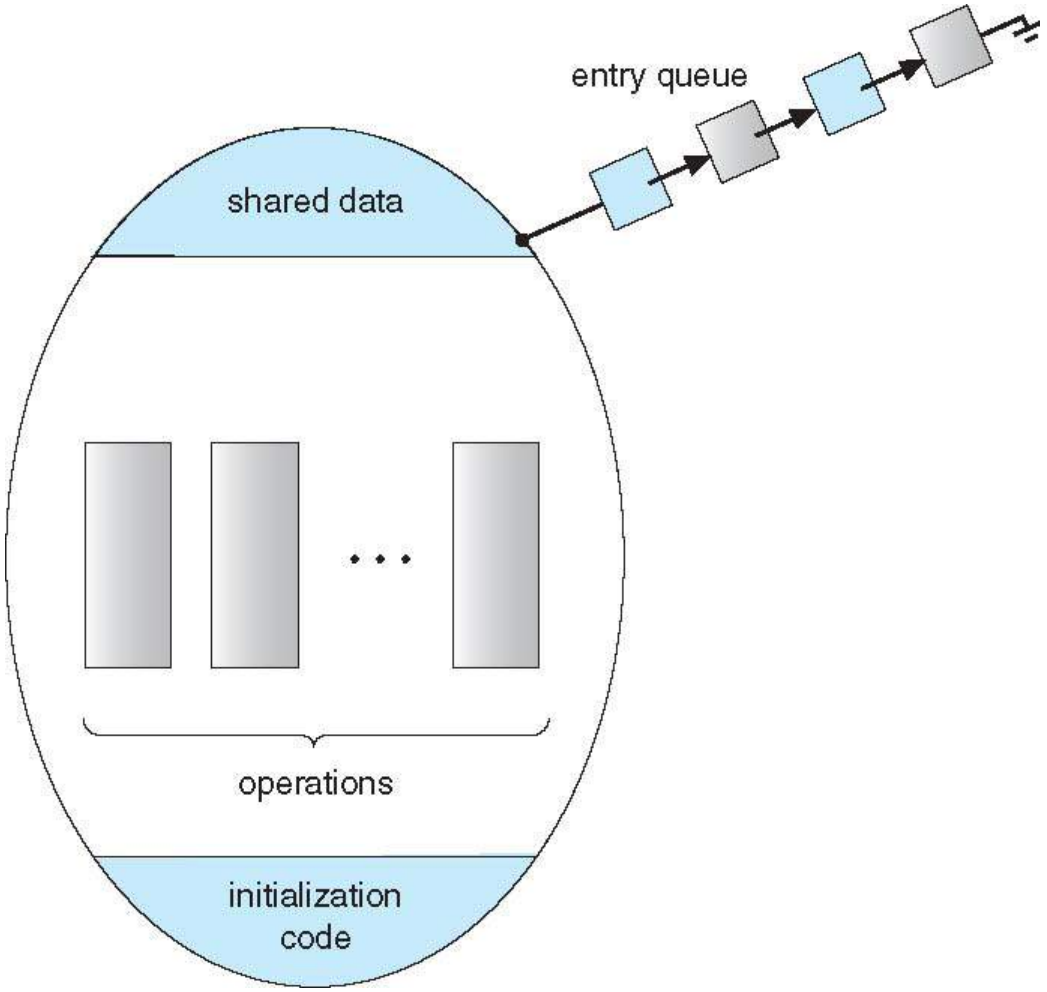
- Incorrect use of semaphore operations:
  - signal (mutex) …. wait (mutex)
  - wait (mutex) … wait (mutex)
  - Omitting of wait (mutex) or signal (mutex) (or both)
- Deadlock and starvation are possible.

Monitors

- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- *Abstract data type*, internal variables only accessible by code within the procedure
- Only one process may be active within the monitor at a time
- But not powerful enough to model some synchronization schemes

```
monitor monitor-name
{
  // shared variable declarations
  procedure P1 (…) { …. }

  procedure Pn (…) {……}

     Initialization code (…) { … }
  }
}
```

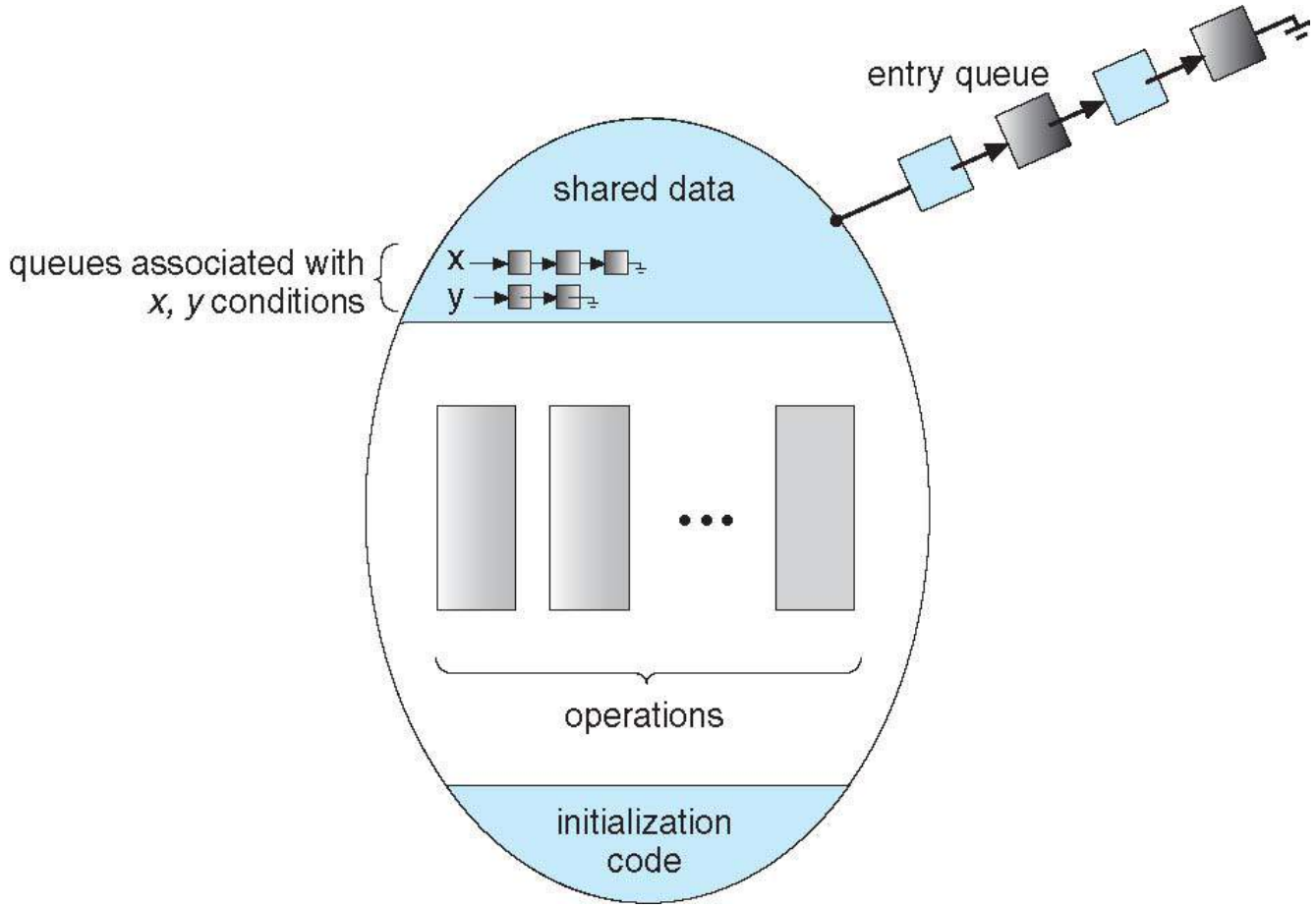
Schematic view of a Monitor



Condition Variables

- `condition x, y;`
- Two operations are allowed on a condition variable:
  - **`x.wait()`** – a process that invokes the operation is suspended until **`x.signal()`**
  - **`x.signal()`** – resumes one of processes (if any) that invoked **`x.wait()`**
    - If no **`x.wait()`** on the variable, then it has no effect on the variable

Monitor with Condition Variables



Condition Variables Choices

- If process P invokes **`x.signal()`**, and process Q is suspended in **`x.wait()`**, what should happen next?
  - Both Q and P cannot execute in paralel. If Q is resumed, then P must wait
- Options include
  - **Signal and wait** – P waits until Q either leaves the monitor or it waits for another condition
  - **Signal and continue** – Q waits until P either leaves the monitor or it waits for another condition
  - Both have pros and cons – language implementer can decide
  - Monitors implemented in Concurrent Pascal compromise
    - P executing signal immediately leaves the monitor, Q is resumed
  - Implemented in other languages including Mesa, C#, Java

Solution to Dining Philosophers

- Each philosopher *i {0 to 4}* invokes the operations **`pickup()`** and **`putdown()`** in the following sequence:

```
DiningPhilosophers.pickup(i);

    EAT

DiningPhilosophers.putdown(i);
```

- No deadlock, but starvation is possible

Solution to Dining Philosophers (Cont.)

```
monitor DiningPhilosophers
{
   enum{THINKING; HUNGRY; EATING} state [5] ;
   condition self [5];

   void pickup (int i) {
          state[i] = HUNGRY;
          test(i);
          if (state[i] != EATING) self[i].wait;
    }

   void putdown (int i) {
          state[i] = THINKING;
                  // test left and right neighbors
           test((i + 4) % 5);
           test((i + 1) % 5);
    }
```

Solution to Dining Philosophers (Cont.)

```
    void test (int i) {
            if ((state[(i + 4) % 5] != EATING) &&
            (state[i] == HUNGRY) &&
            (state[(i + 1) % 5] != EATING)) {
                state[i] = EATING ;
             self[i].signal() ;
            }
    }

    initialization_code() {
           for (int i = 0; i < 5; i++)
           state[i] = THINKING;
    }
}
```

Monitor Implementation Using Semaphores

- Variables

```
semaphore mutex;  // (initially  = 1)
semaphore next;   // (initially  = 0)
int next_count = 0;
```

- Each procedure ***F*** will be replaced by

```
wait(mutex);
      …
   body of F;
      …
if (next_count > 0)
  signal(next)
else
  signal(mutex);
```

- Mutual exclusion within a monitor is ensured

Resuming Processes within a Monitor

- If several processes queued on condition x, and x.signal() executed, which should be resumed?
- FCFS frequently not adequate
- **conditional-wait** construct of the form x.wait(c)
  - Where c is **priority number**
  - Process with lowest number (highest priority) is scheduled next

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

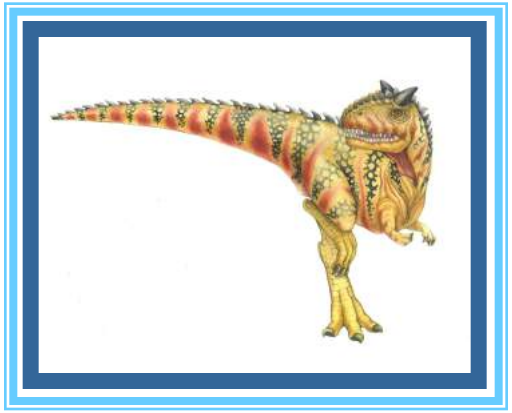
Pthreads Synchronization

- Pthreads API is OS-independent
- It provides:
  - mutex locks
  - condition variable
- Non-portable extensions include:
  - read-write locks
  - spinlocks

Alternative Approaches

- Transactional Memory
- OpenMP
- Functional Programming Languages

End of Chapter 5



Threads and terminology

- A reentrant method is one that can be called simultaneously by multiple threads, provided no two invocations of the method attempt to reference the same data.

?

```
static int sum = 0;

int increment(int i) {
  sum += i;
  return sum;
}
```

?

```
int increment(int sum, int i) {
    return sum + i;
}
```

Threads and terminology

- A reentrant method is one that can be called simultaneously by multiple threads, provided no two invocations of the method attempt to reference the same data.

Non-Reentrant

```
static int sum = 0;

int increment(int i) {
  sum += i;
  return sum;
}
```

Reentrant

```
int increment(int sum, int i) {
    return sum + i;
}
```

Threads and terminology

- A thread-safe method can be called simultaneously by multiple threads at any time, because any shared data is protected somehow (e.g., by a mutex) from simultaneous accesses.
- A thread-safe object is one that can be accessed concurrently by multiple threads and is guaranteed to always be in a "valid" state.

QT Processes

- QProcess is the class for starting and controlling other processes.
- A QProcess can launch another process using the *start()* function.
  - The new process is a child process that terminates when the parent process does.

Example of QProcess

```
LogTail::LogTail(QString fn) {
    ...
    QStringList argv;

    argv << "-f" << fn;
    start("tail", argv);
}
```

Qt Thread (or QThread)

- Qt's thread model permits the prioritizing and control of threads.
- One important class in this namespace is QThreadPool, a class that manages a pool of threads.
  - Every Qt application has a QThreadPool with a suggested maximum thread count that defaults, on most systems, to the number of cores.
- Qt uses a Message Queue to pass signals between threads.

How Qthread works

- Qthread start() method creates a new thread, which will execute the run() method and ends when it finishes

```
class MyThread : public QThread {
    Q_OBJECT
    protected: void run();
};

void MyThread::run() { ... }
```

Synchronizing Threads

- In Qt, the following mechanisms are available:
  - Qmutex
  - QReadWriteLock
  - QSemaphore
  - QWaitCondition

QMutex

- Provides a Mutual exclusive lock for shared data
- Methods:
  - lock
  - tryLock(int timeout)
    - Attempts to lock the mutex. This function returns true if the lock was obtained; otherwise it returns false. If another thread has locked the mutex, this function will wait for at most timeout milliseconds for the mutex to become available.
  - unlock
- If the mutex is already locked, any other thread trying to lock it will sleep until it is unlocked
- When the mutex is unlocked, a single thread that is blocked on lock will be released and start its work

QReadWriteLock

- Works a bit like QMutex but differentiates read and write operations
- Allows multiple readers and a single writer
  - *"readers attempting to obtain a lock will not succeed if there is a blocked writer waiting for access, even if the lock is currently only accessed by other readers. Also, if the lock is accessed by a writer and another writer comes in, that writer will have priority over any readers that might also be waiting."*
- Methods:
  - lockForRead
  - lockForWrite
  - try...
  - unlock

QSemaphore

- A general purpose counting semaphore
- Methods:
  - acquire(int n=1)
  - release(int n=1)
  - int available()

QWaitCondition

- A general purpose signal/wait mechanism
  - Basically a conditional variable
- Allows a thread to tell another thread that a condition has been met, and it is now safe to resume work
- Perfect for Producer/Consumer
- Methods:
  - wait
  - wakeAll()
  - wakeOne()

Example of QWaitCondition

- *keyPressed* variable is a global variable of type QWaitCondition.

Thread 1

```
…
keyPressed.wait(&mutex);
    do_something();
…
…
```

Thread 2

```
…
    getchar();
keyPressed.wakeAll();
…
```

A common design pattern to avoid direct use of threads

- An event-loop is a well-known design pattern to produce multi-threaded code without managing threads explicitly

```
while (is_active)
{
    while (!event_queue_is_empty)
        dispatch_next_event();

}
```

Event Loop instead of threading

- A dispatcher is in charge of activating a new thread (usually from a pool) once a new event comes
  - Event can be handled blocking or non-blocking
- An event may be produced as a consequence of:
  - Timer expiration (Qtimer)
  - User-defined signal that identifies the arrival of a new even

Even loop Scheme