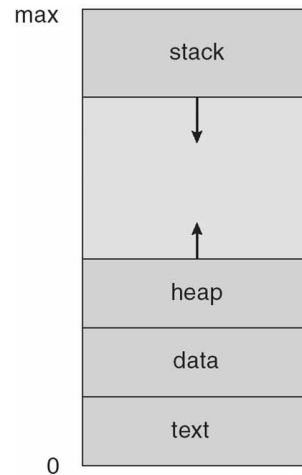
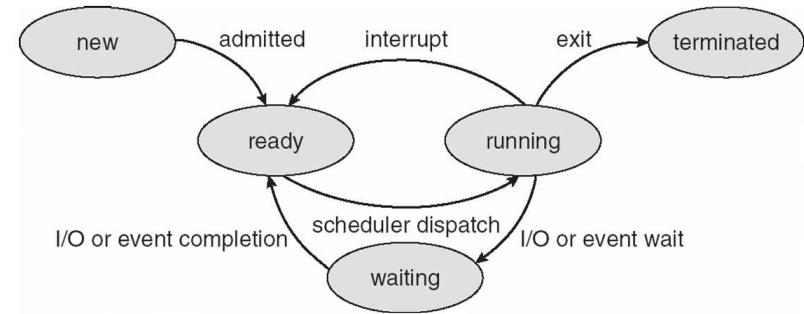


Processes

- The fundamental unit of computation: process
 - Contains an address space and one or more threads of execution (program counters)
- A process includes:
 - program counter(s)
 - stack
 - heap
 - data section
- As a process executes, it changes state
 - new: the process is being created
 - running: instructions are being executed
 - waiting: waiting for some event to occur
 - ready: waiting to be assigned to a processor
 - terminated: finished execution

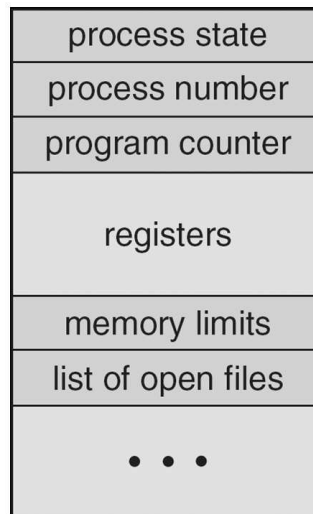


Process State

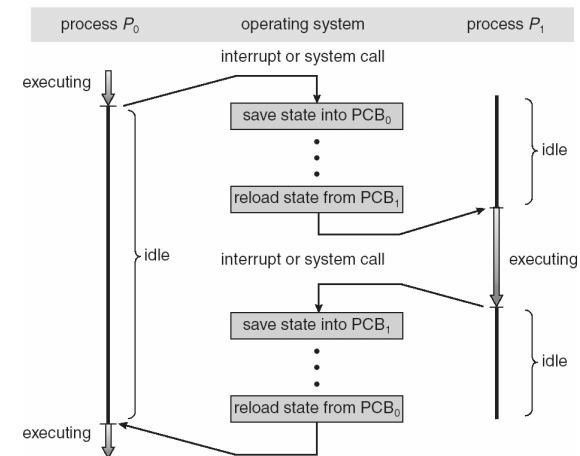


Process Control Block (PCB)

- Information associated with each process
 - Process state
 - Program counter
 - CPU registers
 - CPU scheduling information
 - Memory-management information
 - Accounting information
 - I/O status information



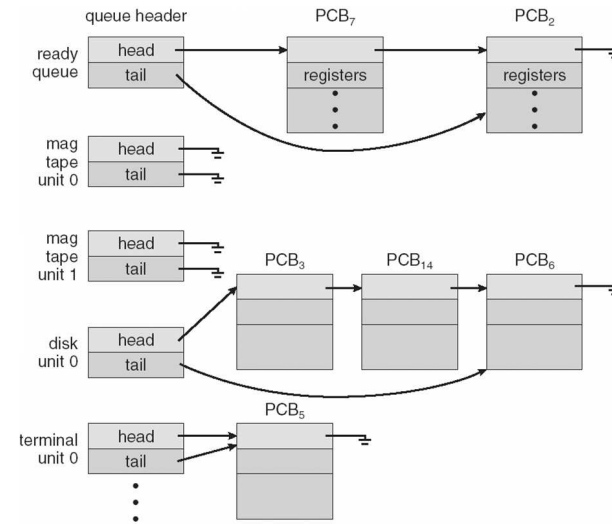
CPU Switch From Process to Process



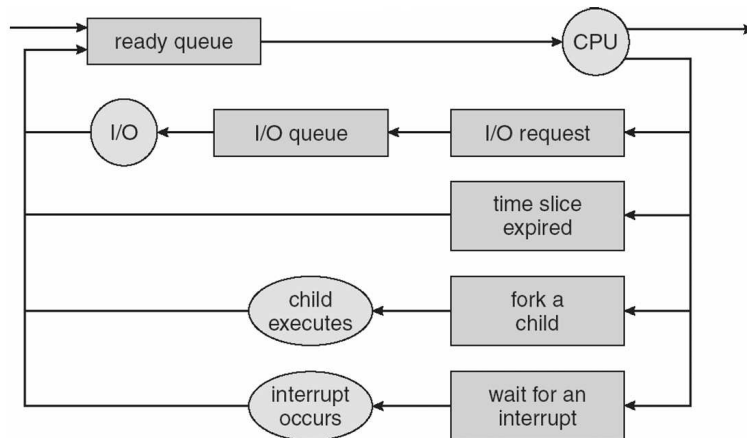
Scheduling Queues

- 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 I/O Device Queues



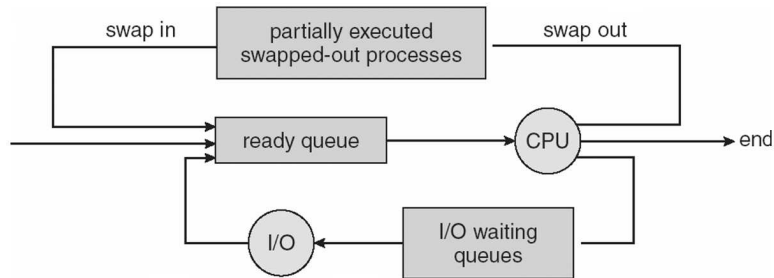
Process Scheduling



Schedulers

- Long-term scheduler (or job scheduler) – selects which processes should be brought into the ready queue
- Short-term scheduler (or CPU scheduler) – selects which process should be executed next and allocates CPU
- Short-term scheduler is invoked very frequently (milliseconds) - must be fast
- 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; fewer, longer CPU bursts
- Scheduling is based on context switching

Medium Term Scheduling

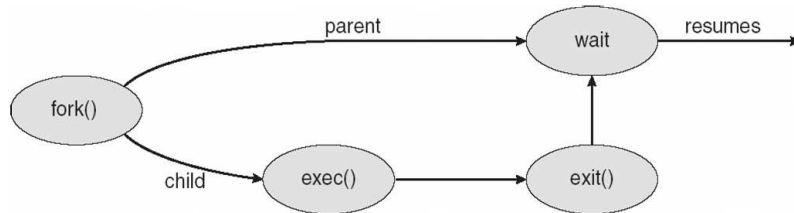


Context Switching

- When the 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
- Time dependent on hardware support

Process Creation

- A parent process create children processes, which, in turn may create other processes
 - We have a tree of processes
- Processes are identified and managed via a process identifier (pid)
- Resource sharing: share all resources / share a subset of resources / share no resource
- Execution: parent and child execute concurrently / parent waits until child terminates
- Address space: child duplicates the parent / child has a program loaded into it
- UNIX examples
 - `fork()` creates new process by duplicating the parent
 - `exec()` replace the process' memory space with a new program (usually used after `fork`)



Creating processes

```
#include <iostream.h>
#include <sys/types.h>
#include <unistd.h>
int main (int argc, char** argv) {
    int i;
    int sum = 0;
    fork();
    for (int i=1; i<10000; i++) {
        sum = sum + i;
    }
    printf("\nI computed %d\n", sum);
}
```

Creating processes (cont'd)

- The call to `fork()` duplicates the current process; both processes continue execution from the instruction following the call to `fork()`

- The initial process is the parent, and the newly created copy is called a child

- What if we want the two processes to do different things?

- `fork()` returns two different integers in the child and parent processes

- In the child process `fork()` returns zero

- In the parent process `fork()` returns the PID of the newly created child

- So we provide different code for the parent and the child, and we surround them by appropriate `if` statements

```
int main (int argc, char** argv) {
    int i;
    int sum = 0;
    int pid;
    pid = fork();
    for (int i=1; i<10000; i++) {
        if (pid == 0)
            cout << "+"; // child process
        else
            cout << "-"; // parent process
        cout.flush();
        sum = sum + i; // both processes
    }
    if (pid == 0)
        cout << "\n[Child] I computed "
            << sum << "\n";
    else
        cout << "\n[Parent] I computed "
            << sum << "\n";
}
```

Creating processes (cont'd)

- The call `execve` replaces completely the current process with another executable.
 - The arguments are the name of the command to execute, then two arrays of strings containing the command line arguments and the environment, just as the ones received by the function `main`.
- Suppose now that we want to run an external command (so we use `execve`)...
- ... but we want to continue the execution of the main program too.

Creating processes (cont'd)

- The call `execve` replaces completely the current process with another executable.
 - The arguments are the name of the command to execute, then two arrays of strings containing the command line arguments and the environment, just as the ones received by the function `main`.
- Suppose now that we want to run an external command (so we use `execve`)...
- ... but we want to continue the execution of the main program too.

We use a combination of `fork` and `execve`:

```
int childp = fork();
if (childp == 0) { // child
    execve(command, argv, envp);
}
else { // parent
    // code that continues our program
}
```

Know What Your Children Do

Again, suppose that we want to execute an external command (`execve` again), we still want to continue the execution of the main program, but only after the external command has been completed.

Know What Your Children Do

Again, suppose that we want to execute an external command (execve again), we still want to continue the execution of the main program, but only after the external command has been completed.

```
int run_it (char* command, char* argv [], char* envp[]) {
    int childp = fork();
    int status;
    if (childp == 0) { // child
        execve(command, argv, envp);
        perror("run_it");
        exit(1);
    }
    else { // parent
        waitpid(childp, &status, 0);
    }
    return status;
}
```

Variants of `waitpid` and `execve` also exist

Process Termination (cont'd)

- Issue: If we use `fork` to create new processes, they will not terminate completely (zombie processes).
 - A zombie process dies for good when its parent executes `waitpid` on them.
 - When a child process terminates, it sends a `SIGCHLD` signal to its parent.
 - Ergo, we can create a function `zombie_reaper` that fires up whenever a `SIGCHLD` signal is received:
 - We then put before the call to `fork`:

```
signal(SIGCHLD, zombie_reaper);
```

```
void zombie_reaper (int sig) {
    int status;
    while (waitpid(-1, &status, WNOHANG) >= 0) ; //nop
}
```

Process Termination

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

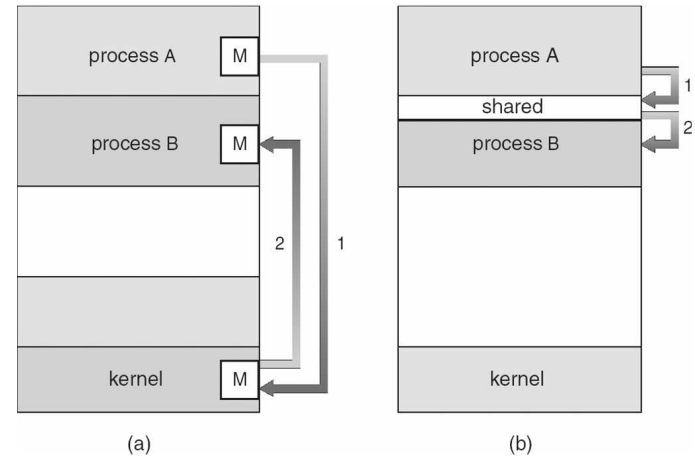
Intermission: Signals

- Processes can send signals to each other
 - A signal is just an `int`
 - It is sent automatically by the system in some cases (e.g., `SIGCHLD` to the parent when the child terminates)
 - But you can also send signals from your code (the meaning of most signals is defined by the system, but `SIGUSR1` and `SIGUSR2` are user-defined)
 - For a description of available signals, see `man -S7 signal`
- To send a signal, use the function `int kill(pid_t pid, int sig);` (see `man -S2 kill`) where `pid` is
 - the process id of some process, or
 - 0 to send the signal to all the processes in the process group
 - -1 to send the signal to all the processes except the first one (init)
- To receive a signal, you should establish a signal handler in your program by using the function `signal` (see `man -S2 signal`)
 - The signal handler fires asynchronously once for each received signal
 - Special signal that cannot be handled: `SIGKILL`

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



The Producer-Consumer Paradigm (Shared Memory)

- Paradigm for cooperating processes, producer process produces information that is consumed by a consumer process
- Communication is accomplished via a buffer
 - unbounded-buffer places no practical limit on the size of the buffer
 - bounded-buffer assumes that there is a fixed buffer size

```

while (true) { /* producer */
    while ((in = (in + 1) % BUFFER_SIZE) == out)
        ; // no buffer available
    buffer[in] = item; in = (in + 1) % BUFFER_SIZE; }
while (true) { /* consumer */
    while (in == out) ; // nothing to consume
    // remove an item from the buffer
    item = buffer[out];
    out = (out + 1) % BUFFER_SIZE; return item; }

```

```

#define BSIZE 10
typedef struct {
    . . .
} item;
item buffer[BSIZE];
int in = 0;
int out = 0;

```

Interprocess Communication – Message Passing

- Mechanism for processes to communicate and to synchronize their actions
- Message system – processes communicate without resorting to shared variables
- IPC facility provides two operations:
 - **send**(message) – message size fixed or variable
 - **receive**(message)
- If *P* and *Q* wish to communicate, they need to:
 - establish a communication link between them
 - exchange messages via send/receive
- Implementation of communication link
 - physical (e.g., shared memory, hardware bus)
 - logical (e.g., logical properties)

Possible Implementations

- Direct communication: Processes name each other explicitly
 - **send** (P , *message*) / **receive** (Q , *message*)
 - 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 (or ports)
 - Each mailbox has a unique id
 - Processes can communicate only if they share a mailbox
 - 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 (cont'd)

- Operations
 - create and destroy a new mailbox
 - send and receive messages through mailbox
- Sending and receiving operations now defined as:
 - **send** (A , *message*) / **receive** (A , *message*) – for some mailbox A
- 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 and Buffering

- Message passing may be either blocking or non-blocking
- Blocking or synchronous:
 - Blocking send has the sender block until the message is received
 - Blocking receive has the receiver block until a message is available
- Non-blocking or asynchronous:
 - Non-blocking send has the sender send the message and continue
 - Non-blocking receive has the receiver receive a valid message or not
- Buffering implies the queuing of messages attached to the link
 1. Zero capacity – 0 messages
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

Shared Memory IPC: POSIX

POSIX Shared Memory

- › Process first creates shared memory segment

```
segment_id = shmget(IPC_PRIVATE, size, S_IRUSR | S_IWUSR);
```
- › Process wanting access to that shared memory must attach to it

```
shared_memory = (char *) shmat(id, NULL, 0);
```
- › Now the process could write to the shared memory

```
sprintf(shared_memory, "Writing to shared memory");
```
- › When done a process can detach the shared memory from its address space

```
shmdt(shared_memory);
```

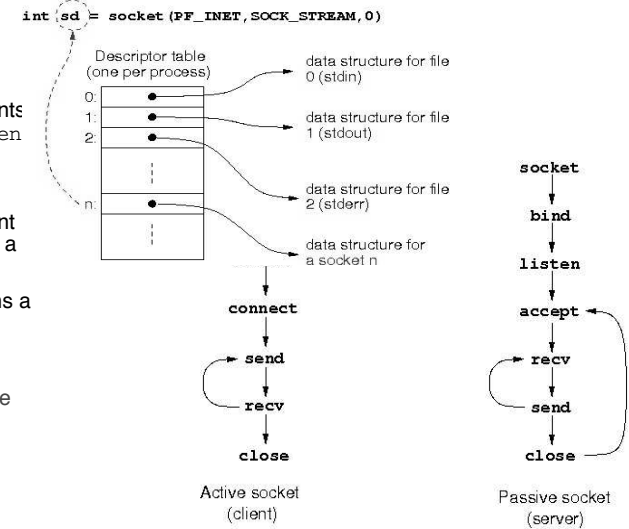
Direct Communication IPC: UNIX Pipes

- Shared memory is handy but poses a series of risks of data corruption
- Message passing (of any kind) is usually preferred
- One of the oldest and most elegant communication system is the pipe
 - Fully buffered stream paradigm
 - The system call `pipe` produces two file descriptors

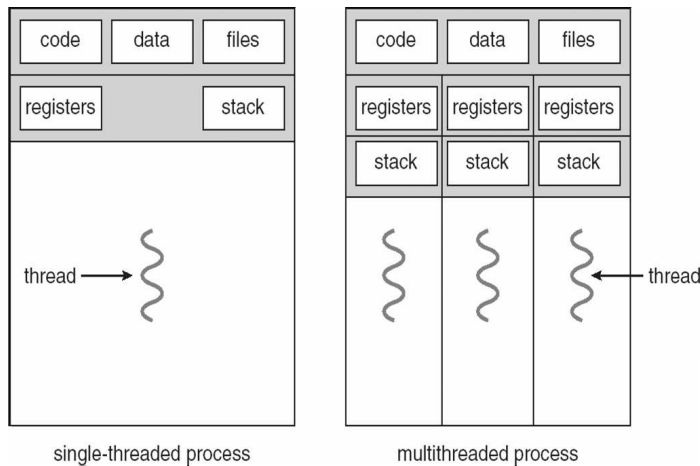

```
char com[2]; r = pipe (com);
```
 - Everything written into `com[1]` can be read back from `com[0]` in the same order

Mailbox Communication IPC: Berkley Sockets

- Used especially in network communication (but can also be used locally)
- An application that wants to access a file uses `open`
 - `open` returns a file descriptor.
- An application that want to communicate creates a socket using `socket`.
 - `socket` also returns a descriptor.
- We can then convince the socket to be either passive (server) or active (client)

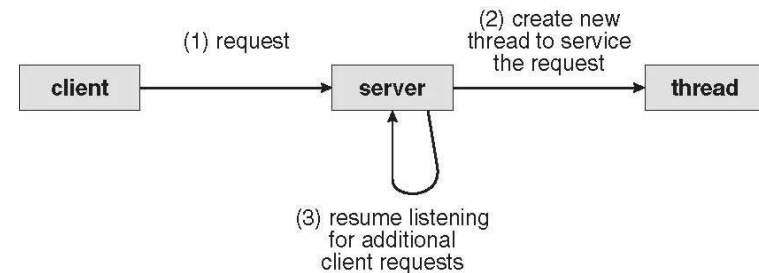


Multithreaded Processes



Benefits

- Responsiveness, resource sharing, economy, scalability
- Shared memory very handy
- Critical on systems where creating processes is expensive
- Possible xample: multi-threaded servers

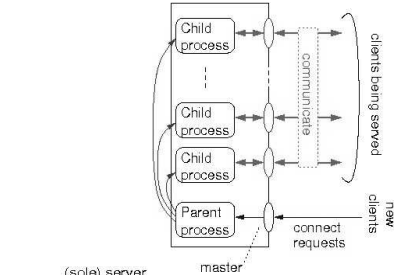


Downsides

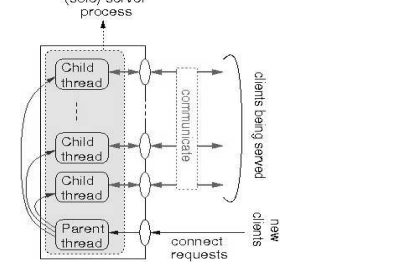
- Main downside: thread interference
 - System calls in particular may not be thread safe
 - Most are, but thread safeness is not always documented
- Lack of robustness: if a thread performs an illegal operation (e.g., access an illegal memory address) the whole process gets killed.
- Other caveats:
 - File and socket descriptors are shared
 - Once a thread opens a file/socket, it is opened for all the threads
 - Most importantly, once a thread closes a descriptor, no other thread can access that descriptor successfully
 - If a thread calls `exit` then the whole process terminates
 - A thread terminates itself when the top-level function (i.e., the function that created it) returns, or when the main function of the thread returns, or explicitly by calling `pthread_exit`

Threads vs. Processes

place in passive mode the master socket
 repeat forever:
 accept the next connection request
 create a new slave socket `s`
 fork
 in children process
 close master socket
 serve requests from the client (via `s`)
 in parent process
 close slave socket



place in passive mode the master socket
 repeat forever:
 accept the next connection request
 create a new slave socket `s`
 in a new thread:
 do not close master socket
 serve requests from the client (via `s`)
 in parent thread
 do not close slave socket

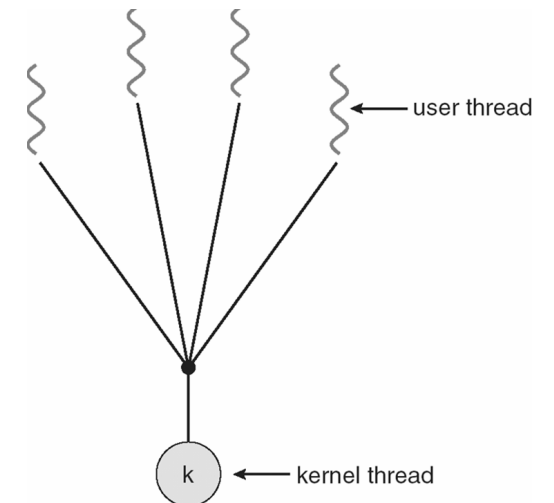


Threads

- User threads:
 - Thread management done by user-level threads library
 - Three primary thread libraries: POSIX Pthreads, Win32 threads, Java threads
- Kernel threads:
 - Supported by the Kernel
 - Examples include Windows XP/2000, Linux, Mac OS, etc.
- Multithreading models: many-to-one, one-to-one, many-to-many

Many-to-One Model

Solaris Green Threads
 GNU Portable Threads

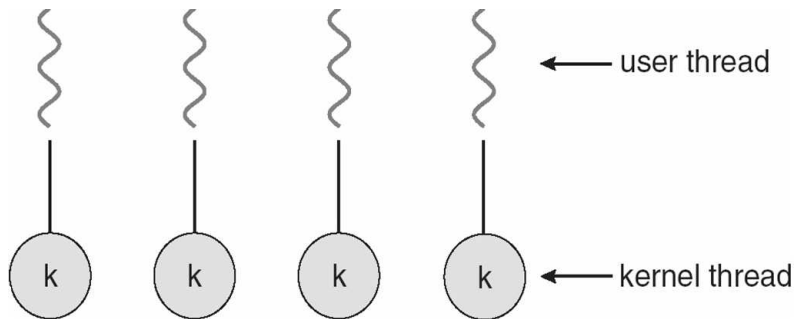


One-to-one Model

Windows NT/XP/2000

Linux

Solaris 9 and later

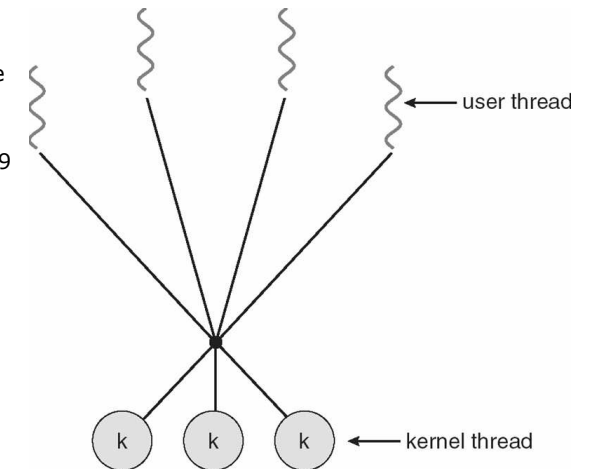


CSC 309, Winter 2010

Processes/37

Many-to-Many Model

- Allows many user level threads to be mapped to many kernel threads
- Allows the OS to create a sufficient number of kernel threads
- Solaris prior to version 9
- Windows NT/2000 with the *ThreadFiber* package



CSC 309, Winter 2010

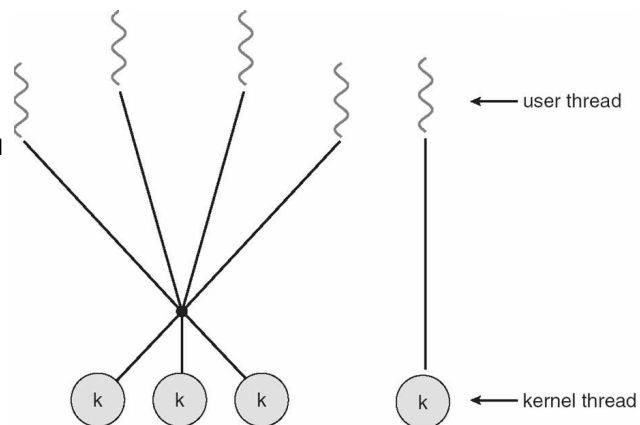
Processes/38

Two-level Model

• Similar to many-to-many, except that it allows a user thread to be bound to kernel thread

• Examples include

- IRIX
- HP-UX
- Tru64 UNIX
- Solaris 8 and earlier



CSC 309, Winter 2010

Processes/39

Thread Libraries

- Thread libraries 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
 - POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
 - API specifies behaviour, implementation is up to development of the library
 - Common in UNIX operating systems (Solaris, Linux, Mac OS X)

CSC 309, Winter 2010

Processes/40

Threading Issues

- Semantics of `fork()` and `exec()` system calls
 - Does `fork()` duplicate only the calling thread or all threads?
- Thread cancellation of target thread (terminating a thread before it has finished)
- Two general approaches:
 - Asynchronous cancellation terminate immediately
 - Deferred cancellation thread checks periodically if it should be cancelled
- Signal handling
- Thread pools: Create a number of threads in a pool where they await work
 - Usually slightly faster to service a request with an existing thread
 - Allows the number of threads to be bound to the size of the pool
- Thread-specific data
 - Allows each thread to have its own copy of data. Useful when one does not have control over the thread creation (e.g., thread pool)
- Scheduler activations

Scheduler Activations

- Both many-to-many and two-level models require communication to maintain the appropriate number of kernel threads allocated to the application
- Scheduler activations provide upcalls - a communication mechanism from the kernel to the thread library
- This communication allows an application to maintain the correct number kernel 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
- Options:
 - 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

Pthreads (cont'd)

- Threads are created at any time using the system call `pthread_create`
- Threads execute concurrently and are preemptible (one thread cannot block the CPU)
 - A thread can give up the CPU voluntarily by using the system call `sched_yield` (also available for processes)
- The threads API include functions for coordination and synchronization
- A program that uses threads must include `pthread.h` and must be linked with the library `pthread`, i.e.,

```
g++ -lpthread -o foo foo.cc
g++ -lpthread -o tserv tserv.o tcp-utils.o
```

Termination and Cancellation

- A thread can terminate by returning from its main function or by calling `pthread_exit`
- A thread can terminate others by sending a cancellation request using `pthread_cancel`
- Sole argument: the thread being cancelled (`pthread_t`)
- Depending on its settings, the target thread can ignore the request, honor it immediately, or defer it till it reaches a cancellation point
- The following POSIX threads functions are cancellation points:
 - `pthread_join`, `pthread_cond_wait`, `pthread_cond_timedwait`, `pthread_testcancel`, `sem_wait`, `sigwait`
- All other POSIX threads functions are guaranteed not to be cancellation point
- `pthread_testcancel` tests for pending cancellation and executing it
- When the cancellation is honored the thread being cancelled behaves as if it calls `pthread_exit(PTHREAD_CANCELED)`
- In addition to the cancellation points enumerated above, a number of system calls (basically, all system calls that may block) and library functions that may call these system calls are cancellation points
- However, older implementations do not conform to this; workaround:
 - Cancellation requests are transmitted to the target thread by sending it a signal
 - The signal will interrupt all blocking system calls, causing them to return with `EINTR`
 - So `pthread_cancel` immediately after a system call is safe and achieves the desired effect

Linux Threads

- Linux refers to them as tasks rather than threads
- Thread creation is done through `clone()` system call
- `clone()` allows a child task to share the address space of the parent task (process)

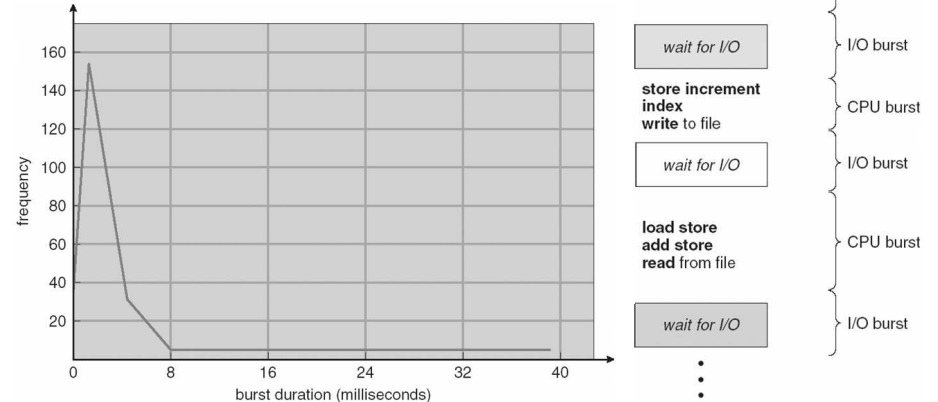
flag	meaning
<code>CLONE_FS</code>	File-system information is shared.
<code>CLONE_VM</code>	The same memory space is shared.
<code>CLONE_SIGHAND</code>	Signal handlers are shared.
<code>CLONE_FILES</code>	The set of open files is shared.

Cancellation Points

- `pthread_setcancelstate` changes the cancellation state for the calling thread
 - that is, whether cancellation requests are ignored or not (possible state values: `PTHREAD_CANCEL_DISABLE`, `PTHREAD_CANCEL_ENABLE`)
 - the old cancellation state is stored and can thus be restored (unless the second argument is 0)
 - `pthread_setcanceltype` changes the type of responses to cancellation requests
 - possible behaviour: asynchronous (immediate) or deferred cancellation (`PTHREAD_CANCEL_ASYNCHRONOUS`, `PTHREAD_CANCEL_DEFERRED`)
 - the old cancellation type is stored and can thus be restored (unless the second argument is 0)
- A thread is created by default with cancellation enabled and deferred

CPU Scheduling

- Maximum CPU utilization via multiprogramming
- CPU-I/O Burst Cycle: Process execution consists of a cycle of CPU execution and I/O wait
- CPU burst distribution



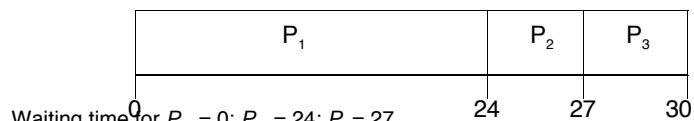
The CPU Scheduler

- Selects from among the processes in memory that are ready to execute, and allocates the CPU to one of them
 - CPU scheduling decisions may take place when a process:
 1. Switches from running to waiting state
 2. Switches from running to ready state
 3. Switches from waiting to ready
 4. Terminates
 - Scheduling under 1 and 4 is nonpreemptive
 - All other scheduling is preemptive
 - Dispatcher module gives control of the CPU to the process selected by the short-term scheduler; this involves:
 - switching context
 - switching to user mode
 - jumping to the proper location in the user program to restart that program
- Dispatch latency – time it takes for the dispatcher to stop one process and start another

First-Come, First-Served (FCFS) Scheduling

Process	Burst Time
P_1	24
P_2	3
P_3	3

Suppose that the processes arrive in the order: P_1, P_2, P_3



Average waiting time: $(0 + 24 + 27)/3 = 17$

Scheduling Criteria

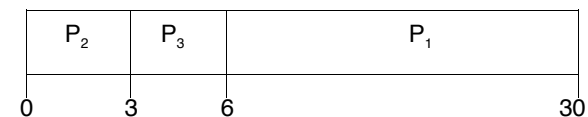
- CPU utilization – keep the CPU as busy as possible
- Throughput – # of processes that complete their execution per time unit
- Turnaround time – amount of time to execute a particular process
- Waiting time – amount of time a process has been waiting in the ready queue
- Response time – amount of time it takes from when a request was submitted until the first response is produced (not output!)

FCFS Scheduling (cont'd)

Suppose that the processes arrive in the order

P_2, P_3, P_1

The Gantt chart for the schedule is:



Waiting time for $P_1 = 6; P_2 = 0; P_3 = 3$

Average waiting time: $(6 + 0 + 3)/3 = 3$

Much better than previous case

Convoy effect -- short process behind long process

Shortest-Job-First (SJF) Scheduling

- Associate with each process the length of its next CPU burst.
- Use these lengths to schedule the process with the shortest time
- SJF is optimal – gives minimum average waiting time for a given set of processes
- The difficulty is knowing the length of the next CPU request

Process	Arrival Time	Burst Time
P_1	0.0	6
P_2	2.0	8
P_3	4.0	7
P_4	5.0	3

0 3 9 16 24

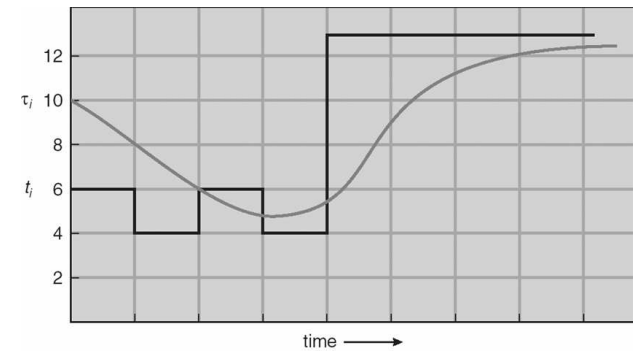
$$\text{Average waiting time} = (3 + 16 + 9 + 0) / 4 = 7$$

Priority Scheduling

- A priority (integer) is associated with each process
- CPU is allocated to the process with the highest priority (smallest integer = highest priority)
 - Preemptive
 - Nonpreemptive
- SJF is a priority scheduling where priority is the predicted next CPU burst time
- Problem: Starvation – low priority processes may never execute
- Solution: Aging – as time progresses increase the priority of the process

Determining Length of Next CPU Burst

- Can only estimate the length
- Can be done by using the length of previous CPU bursts, using exponential averaging



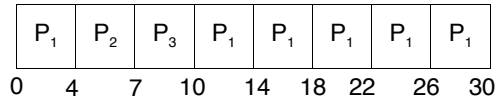
CPU burst (t_i)	6	4	6	4	13	13	13	...
"guess" (τ_i)	10	8	6	6	5	9	11	12

Round Robin (RR)

- Each process gets a small unit of CPU time (time quantum), usually 10-100 milliseconds. After this time has elapsed, the process is preempted and added to the end of the ready queue.
- If there are n processes in the ready queue and the time quantum is q , then each process gets $1/n$ of the CPU time in chunks of at most q time units at once. No process waits more than $(n-1)q$ time units.
- Performance
 - q large \rightarrow FIFO
 - q small \rightarrow q must be large with respect to context switch, otherwise overhead is too high

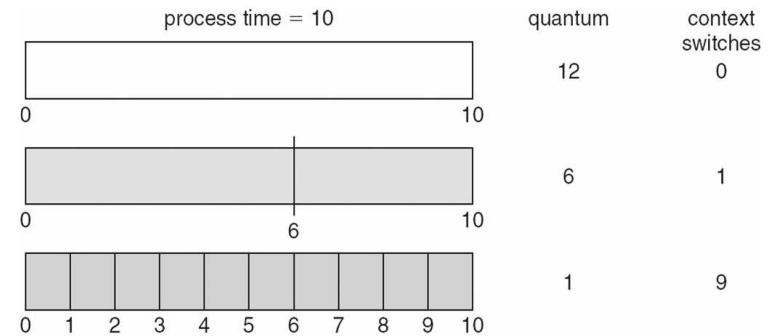
Example of RR (Time Quantum = 4)

Process	Burst Time
P_1	24
P_2	3
P_3	3

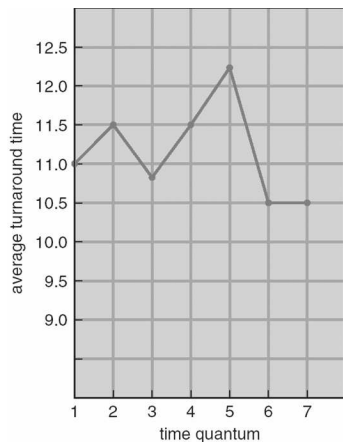


Typically, higher average turnaround than SJF, but better response

Time Quantum and Context Switch Time



Turnaround Time Varies With The Time Quantum

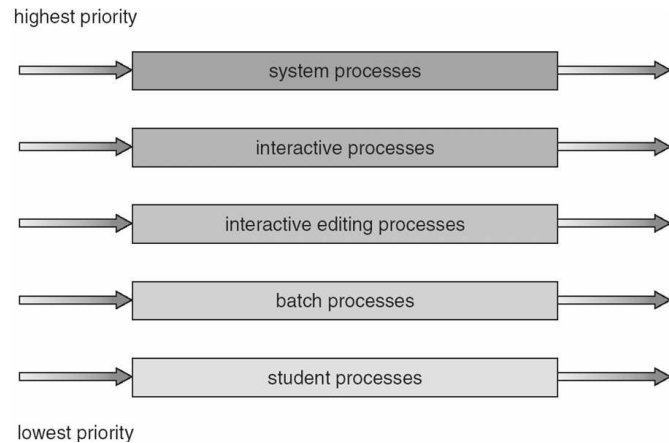


process	time
P_1	6
P_2	3
P_3	1
P_4	7

Multilevel Queue

- Ready queue is partitioned into separate queues:
 - foreground (interactive)
 - background (batch)
- Each queue has its own scheduling algorithm
 - foreground – RR
 - background – FCFS
- Scheduling must be done between the queues
- Fixed priority scheduling; (i.e., serve all from foreground then from background). Possibility of starvation.
- Time slice – each queue gets a certain amount of CPU time which it can schedule amongst its processes; i.e., 80% to foreground in RR, 20% to background in FCFS

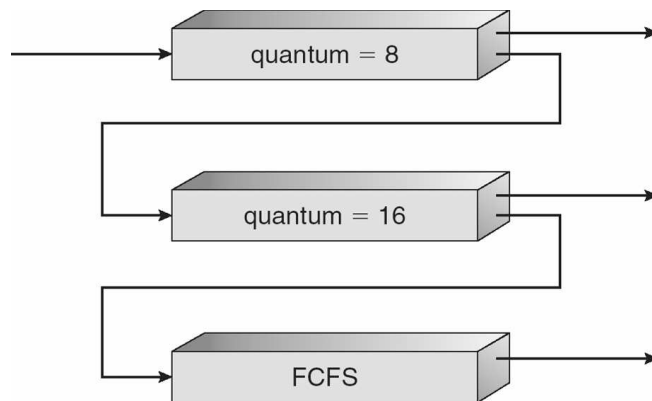
Multilevel Queue Scheduling



Multilevel Feedback Queue

- A process can move between the various queues; aging can be implemented this way
- Multilevel-feedback-queue scheduler defined by the following parameters:
 - number of queues
 - scheduling algorithms for each queue
 - method used to determine when to upgrade a process
 - method used to determine when to demote a process
 - method used to determine which queue a process will enter for the first time
- Example: Q_0 – RR time quantum 8 ms Q_1 – RR time quantum 16 ms Q_2 – FCFS
 - A new job enters queue Q_0 which is served FCFS. When it gains CPU, job receives 8 milliseconds. If it does not finish in 8 milliseconds, job is moved to queue Q_1 .
 - At Q_1 job is again served FCFS and receives 16 additional milliseconds. If it still does not complete, it is preempted and moved to queue Q_2 .

Multilevel Feedback Queues



Thread Scheduling

- Distinction between user-level and kernel-level threads
- 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
- Kernel thread scheduled onto available CPU is system-contention scope (SCS) – competition among all threads in system
- Pthreads:
 - 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.

Pthread Scheduling API

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5
int main(int argc, char *argv[]) {
    int i;    pthread_t tid[NUM_THREADS];    pthread_attr_t attr;
    pthread_attr_t init(&attr); /* get the default attributes */
    /* set the scheduling algorithm to PROCESS or SYSTEM */
    pthread_attr_t setscope(&attr, PTHREAD_SCOPE_SYSTEM);
    /* set the scheduling policy - FIFO, RT, or OTHER */
    pthread_attr_t setschedpolicy(&attr, SCHED_OTHER);
    for (i = 0; i < NUM_THREADS; i++) /* create the threads */
        pthread_create(&tid[i], &attr, runner, NULL);
    for (i = 0; i < NUM_THREADS; i++) /* join on each thread */
        pthread_join(tid[i], NULL);
}
void *runner(void *param) {
    printf("I am a thread\n"); pthread_exit(0);
}
```

CSC 309, Winter 2010

Processes/65

Multiple-Processor Scheduling

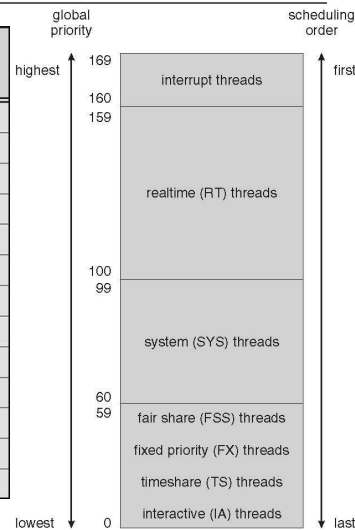
- CPU scheduling more complex when multiple CPUs are available
- Homogeneous processors within a multiprocessor
- Asymmetric multiprocessing – only one processor accesses the system data structures, alleviating the need for data sharing
- Symmetric multiprocessing (SMP) – each processor is self-scheduling, all processes in common ready queue, or each has its own private queue of ready processes
- Processor affinity – process has affinity for processor on which it is currently running
 - soft affinity
 - hard affinity

CSC 309, Winter 2010

Processes/66

Solaris Scheduling

priority	time quantum	time quantum expired	return from sleep
0	200	0	50
5	200	0	50
10	160	0	51
15	160	5	51
20	120	10	52
25	120	15	52
30	80	20	53
35	80	25	54
40	40	30	55
45	40	35	56
50	40	40	58
55	40	45	58
59	20	49	59



CSC 309, Winter 2010

Processes/67

Linux Scheduling

- Constant order $O(1)$ scheduling time
- Two priority ranges: time-sharing and real-time
- Real-time range from 0 to 99 and nice value from 100 to 140

numeric priority	relative priority	time quantum
0	highest	200 ms
•		real-time tasks
•		
99		other tasks
100		
•		
•		
140	lowest	10 ms

CSC 309, Winter 2010

Processes/68

Linux Scheduling (cont'd)



Java Thread Scheduling

- JVM uses a preemptive, priority-based scheduling algorithm
- FIFO queue is used if there are multiple threads with the same priority
- JVM schedules a thread to run when:

1. The currently running thread exits the runnable state
2. A higher priority thread enters the runnable state

The JVM Does Not Specify Whether Threads are Time-Sliced or Not

The `yield()` method should then be used:

```
while (true) {  
    // perform CPU-intensive task  
    . . .  
    Thread.yield();  
}
```

This yields control to another thread of equal priority

Priorities:

- Thread.MIN_PRIORITY
- Thread.MAX_PRIORITY
- Thread.NORM_PRIORITY

```
SetPriority(  
    Thread.NORM_PRIORITY+2);
```

Process Synchronization

- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- Example: 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 `count` that keeps track of the number of full buffers. Initially, `count` 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.

```
while (true) {  
    /* produce an item and put in next */  
    while (count == BUFFER_SIZE); // nop  
    buffer[in] = next;  
    in = (in + 1) % BUFFER_SIZE;  
    count++;  
}
```

```
while (true) {  
    while (count == 0); // nop  
    next = buffer[out];  
    out = (out + 1) % BUFFER_SIZE;  
    count--;  
}
```

Race Condition

- `count++` could be implemented as

```
register1 = count  
register1 = register1 + 1  
count = register1
```

- `count--` could be implemented as

```
register2 = count  
register2 = register2 - 1  
count = register2
```

- Interleaving execution with "count = 5" initially:

```
S0: producer execute register1 = count {register1 = 5}  
S1: producer execute register1 = register1 + 1 {register1 = 6}  
S2: consumer execute register2 = count {register2 = 5}  
S3: consumer execute register2 = register2 - 1 {register2 = 4}  
S4: producer execute count = register1 {count = 6}  
S5: consumer execute count = register2 {count = 4}
```

Problem: Two processes are at the same time in the same critical region (count)

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

Peterson's Solution

- Two process solution
- Assume LOAD and STORE are atomic instructions (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 P_i is ready!

```
repeat { // Process  $P_i$ 
    flag[i] = TRUE;
    turn = j;
    while (flag[j] && turn == j) ;
    critical section
    flag[i] = FALSE;
    remaining code
}
```

Synchronization Hardware

- Many systems provide hardware support for critical section code
- 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-interruptable
 - Either test memory word and set value
 - Or swap contents of two memory words

Solution using Locks, TestAndSet

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

Shared boolean variable lock., initialized to false.

```
repeat { // Process  $P_i$ 
    acquire lock;
    critical section
    release lock;
    remaining code
}

repeat { // Process  $P_i$ 
    while (TestAndSet(&lock)) ;
    critical section
    lock = FALSE;
    remaining code
}
```

Solution using Swap

```
void Swap (boolean *a, boolean *b) {
    boolean temp = *a;
    *a = *b;
    *b = temp;
}
```

Shared Boolean variable lock initialized to FALSE

Each process has a local Boolean variable key

```
repeat { // Process Pi
    key = TRUE;
    while ( key == TRUE) Swap (&lock, &key );
    critical section
    lock = FALSE;
    remaining code
}
```

Bounded-waiting Mutual Exclusion with TestandSet()

```
repeat {
    waiting[i] = TRUE;
    key = TRUE;
    while (waiting[i] && key)
        key = TestAndSet(&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;
    // remaining code
}
```

Pthread Locks

- Realized by mutexes

- **Functions:** pthread_mutex_init, pthread_mutex_lock, pthread_mutex_unlock, pthread_mutex_trylock

```
pthread_mutex_t lock1, lock2;
void* do_lock (int n) {
    pthread_mutex_lock(&lock1);
    cout << "Thread " << n << " enters critical.\n";
    sched_yield(); sleep(3);
    pthread_mutex_unlock(&lock1);
    cout << "Thread " << n << " exits critical.\n";
    return NULL;
}
int main () {
    pthread_mutex_init(&lock1, NULL); pthread_mutex_init(&lock2, NULL);
    pthread_t tt; pthread_attr_t ta; pthread_attr_init(&ta);
    pthread_attr_setdetachstate(&ta, PTHREAD_CREATE_DETACHED);

    pthread_create(&tt, &ta, (void* (*)(void*))do_lock, (void*)1);
    pthread_create(&tt, &ta, (void* (*)(void*))do_lock, (void*)2);
    pthread_create(&tt, &ta, (void* (*)(void*))do_lock, (void*)3);
    sched_yield(); sleep(60);
}
```

Semaphores

- Allow an upper bounded number of threads to be simultaneously in a critical region
 - Semaphore S – integer variable
 - Two standard operations modify S: wait() and signal()
 - Originally called P() and V()
- Can only be accessed via the above two indivisible (atomic) operations

```
wait (S) {
    while (S <= 0)
        ; // no-op
    S--;
}
signal (S) {
    S++;
}
```

Semaphore as General Synchronization Tool

- Counting semaphore – integer value can range over an unrestricted domain
- Binary semaphore – integer value can range only between 0 and 1
 - Generally simpler to implement
 - Also known as mutex locks
- Easy to implement a counting semaphore as a binary semaphore
- Also easy to implement a counting semaphore using a binary semaphore
- Provides mutual exclusion
- Pthread semaphores require the `semaphore.h` header
 - Standard functions: `sem_init`, `sem_wait`, `sem_post`
 - Supplementary functions: `sem_trywait`, `sem_getvalue`

Semaphore Implementation

- Must guarantee that no two processes can execute `wait ()` and `signal ()` on the same semaphore at the same time
- 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
 - 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.

```
wait(semaphore *S) {
    S->value--;
    if (S->value < 0) { add 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

Classical Problems of Synchronization

- The Bounded-Buffer Problem
- The Readers and Writers Problem
- The 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 .

Producer:

```
repeat forever {
  // produce an item in nextp
  wait (empty);
  wait (mutex);
  // add the item to the buffer
  signal (mutex);
  signal (full);
}
```

Consumer:

```
repeat forever {
  wait (full);
  wait (mutex);
  // remove an item from buffer to nextc
  signal (mutex);
  signal (empty);
  // consume the item in nextc
}
```

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
- Shared Data: data set, semaphores `mutex`, `wrt` initialized to 1, integer `readcount` initializes to 0

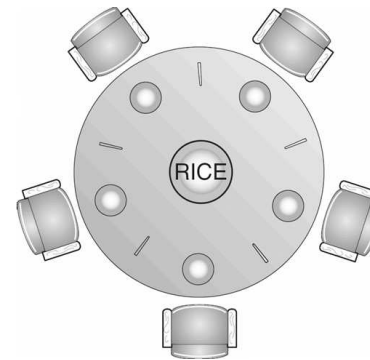
Reader:

```
repeat forever {
  wait (mutex);
  readcount ++;
  if (readcount == 1) wait (wrt);
  signal (mutex);
  // read from dataset
  wait (mutex);
  readcount --;
  if (readcount == 0) signal (wrt);
  signal (mutex);
}
```

Writer:

```
repeat forever {
  wait (wrt);
  // write into data set
  signal (wrt);
}
```

Dining-Philosophers Problem



```
repeat forever {
  wait ( chopstick[i] );
  wait ( chopstick[ (i + 1) % 5] );
  // eat
  signal ( chopstick[i] );
  signal ( chopstick[ (i + 1) % 5] );
  // think
}
```

- Shared data: bowl of rice (data set), semaphores `chopstick` [5] initialized to 1

Monitors

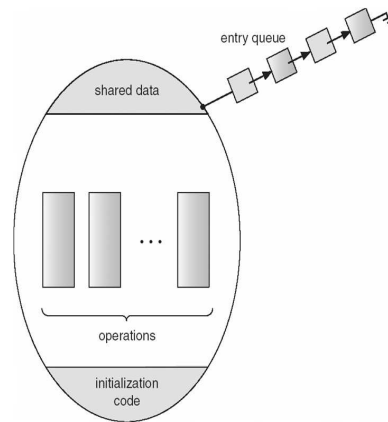
- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- Only one process may be active within the monitor at a time

```

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

    procedure Pn (...) { ..... }

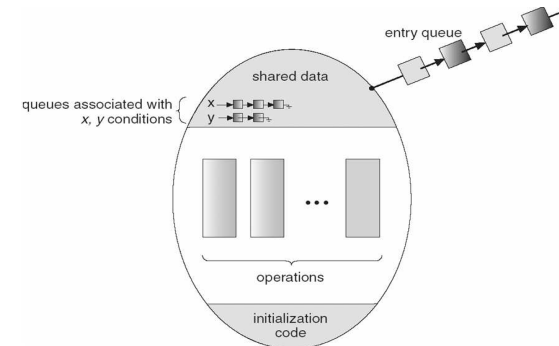
    Initialization code ( .... ) { ... }
    ...
}
    
```



Condition Variables

condition x;

- Two operations on a condition variable:
 - x.wait () – a process that invokes the operation is suspended
 - x.signal () – resumes one of processes (if any) that invoked x.wait ()
- Monitor with condition variables:



Solution to Dining Philosophers

```

monitor DP
{
    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);
    }
}
    
```

Each philosopher invokes the operations pickup() and putdown() in the sequence:

```

pickup (i);
EAT
putdown (i);
    
```

```

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 w/ 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.

Monitor Implementation

For each condition variable x , we have:

```
semaphore x_sem; // (initially = 0)
int x-count = 0;
```

```
x.wait() {
    x-count++;
    if (next_count > 0)
        signal(next);
    else
        signal(mutex);
    wait(x_sem);
    x-count--;
}

x.signal() {
    if (x-count > 0) {
        next_count++;
        signal(x_sem);
        wait(next);
        next_count--;
    }
}
```

A Monitor to Allocate Single Resource

```
monitor ResourceAllocator
{
    boolean busy;
    condition x;
    void acquire(int time) {
        if (busy)
            x.wait(time);
        busy = TRUE;
    }
    void release() {
        busy = FALSE;
        x.signal();
    }
    initialization code() {
        busy = FALSE;
    }
}
```

POSIX Condition Variables

Condition variable = mutex + condition (no direct concept of monitor).

- A number of threads need to access a critical region (mutex).
- Once the critical region is acquired, a certain condition has to be met before going any further.
- While it waits for the condition, a thread gives up the mutex so that other threads may proceed.
- Example: · Wait till x gets larger than y :

```
pthread_mutex_t mut = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
pthread_mutex_lock(&mut);
while (x <= y) {
    pthread_cond_wait(&cond, &mut);
    /* mut is released while waiting */
}
/* mut is reacquired */ /* do stuff with x and y */
pthread_mutex_unlock(&mut);
```

- When x becomes larger than y , the corresponding condition should be signalled:

```
pthread_mutex_lock(&mut);
/* code that changes x and y */
if (x > y) pthread_cond_broadcast(&cond);
pthread_mutex_unlock(&mut);
```

Linux Synchronization

- Prior to kernel Version 2.6, disables interrupts to implement short critical sections
 - Version 2.6 and later, fully preemptive
- Provides: semaphores, spin locks

Atomic Transactions

System Model

Log-based Recovery

Checkpoints

Concurrent Atomic Transactions

Atomic Transactions

- Assures that operations happen as a single logical unit of work, in its entirety, or not at all
 - Related to database systems
- Challenge is assuring atomicity despite computer system failures
- Transaction - collection of instructions or operations that performs single logical function
 - We are concerned with changes to stable storage – disk
 - Transaction is series of read and write operations
 - Terminated by commit (transaction successful) or abort (transaction failed) operation
 - Aborted transaction must be rolled back to undo any changes it performed

Types of Storage Media

- Volatile storage – information stored here does not survive system crashes
 - main memory, cache
- Nonvolatile storage – Information usually survives crashes
 - disk and tape
- Stable storage – Information never lost
 - Not actually possible, so approximated via replication or RAID to devices with independent failure modes
- Goal is to assure transaction atomicity where failures cause loss of information on volatile storage

Log-Based Recovery

- Record to stable storage information about all modifications by a transaction
- Most common: write-ahead logging
- Log on stable storage, each log record describes single transaction write operation, including
 - Transaction name
 - Data item name
 - Old value
 - New value
 - $\langle T_i \text{ starts} \rangle$ written to log when transaction T_i starts
 - $\langle T_i \text{ commits} \rangle$ written when T_i commits
- Log entry must reach stable storage before operation on data occurs
- Using logs the system can handle any volatile memory errors

Log-Based Recovery Algorithm

- Undo(T_i) restores value of all data updated by T_i
- Redo(T_i) sets values of all data in transaction T_i to new values
- Undo(T_i) and redo(T_i) must be idempotent
- Multiple executions must have the same result as one execution
- If system fails, restore state of all updated data via log
 - If log contains $\langle T_i \text{ starts} \rangle$ without $\langle T_i \text{ commits} \rangle$, undo(T_i)
 - If log contains $\langle T_i \text{ starts} \rangle$ and $\langle T_i \text{ commits} \rangle$, redo(T_i)

Checkpoints

- Log could become long, and recovery could take long
- Checkpoints shorten log and recovery time.
- Checkpoint scheme:
 1. Output all log records currently in volatile storage to stable storage
 2. Output all modified data from volatile to stable storage
 3. Output a log record $\langle \text{checkpoint} \rangle$ to the log on stable storage
- Now recovery only includes T_i , such that T_i started executing before the most recent checkpoint, and all transactions after T_i
- All other transactions already on stable storage

Concurrent Transactions

- Must be equivalent to serial execution – serializability
- Could perform all transactions in critical section
- Inefficient, too restrictive
- Concurrency-control algorithms provide serializability
 - Consider two data items A and B
 - Consider Transactions T_0 and T_1
 - Execute T_0 , T_1 atomically
 - Execution sequence called schedule
 - Atomically executed transaction order called serial schedule
 - For N transactions, there are $N!$ valid serial schedules

Nonserial Schedule

- Nonserial schedule allows overlapped execute
 - Resulting execution not necessarily incorrect
- Consider schedule S, operations O_i , O_j
- Conflict if access same data item, with at least one write
- If O_i , O_j consecutive and operations of different transactions & O_i and O_j don't conflict
- Then S' with swapped order O_j , O_i equivalent to S
- If S can become S' via swapping nonconflicting operations
- S is conflict serializable

Concurrent Serializable Schedule

Schedule 1: T₀ then T₁

T ₀	T ₁
read(A)	
write(A)	
read(B)	
write(B)	
	read(A)
	write(A)
	read(B)
	write(B)

Schedule 2: Concurrent serializable

T ₀	T ₁
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

Locking Protocol

- Ensure serializability by associating lock with each data item
- Follow locking protocol for access control
- Locks
 - Shared – T_i has shared-mode lock (S) on item Q, T_i can read Q but not write Q
 - Exclusive – T_i has exclusive-mode lock (X) on Q, T_i can read and write Q
- Require every transaction on item Q acquire appropriate lock
- If lock already held, new request may have to wait
- Similar to readers-writers algorithm
- Two-phase locking:
 - Generally ensures conflict serializability
 - Each transaction issues lock and unlock requests in two phases
 - Growing – obtaining locks
 - Shrinking – releasing locks
 - Does not prevent deadlock

Timestamp-based Protocols

- Select order among transactions in advance – timestamp-ordering
- Transaction T_i associated with timestamp TS(T_i) before T_i starts
- TS(T_i) < TS(T_j) if T_i entered system before T_j
 - TS generated from system clock or as logical counter; TS determine serializability order
 - If TS(T_i) < TS(T_j), system must ensure produced schedule equivalent to serial schedule where T_i appears before T_j
- Implementation:
 - Data item Q gets two timestamps
 - W-timestamp(Q) – largest timestamp of any transaction that executed write(Q) successfully and R-timestamp(Q) – largest timestamp of successful read(Q)
 - Updated whenever read(Q) or write(Q) executed
 - Suppose T_i executes read(Q)
 - If TS(T_i) < W-timestamp(Q), T_i needs to read value of Q that was already overwritten: read operation rejected and T_i rolled back
 - If TS(T_i) ≥ W-timestamp(Q) read executed, R-timestamp(Q) set to max(R-timestamp(Q), TS(T_i))

Timestamp-based Protocols (cont'd)

- Suppose T_i executes write(Q)
- If TS(T_i) < R-timestamp(Q), value Q produced by T_i was needed previously and T_i assumed it would never be produced
- Write operation rejected, T_i rolled back
- If TS(T_i) < W-timestamp(Q), T_i attempting to write obsolete value of Q
- Write operation rejected and T_i rolled back
- Otherwise, write executed
- Any rolled back transaction T_i is assigned new timestamp and restarted
- Algorithm ensures conflict serializability and freedom from deadlock

Schedule Possible w/ Timestamp Protocol

T_2	T_3
read(B)	read(B)
	write(B)
read(A)	read(A)
	write(A)