Chapter 3: Scheduling

Non-preemptive policies: FCFS, SJF Preemptive policies: round robin, multilevel queues w/feedback, guaranteed scheduling Ex. UNIX, Linux, Windows NT and after

The scheduler: part of OS that decides how to allocate the CPU & main memory to processes. Will focus here on the CPU scheduler: decides which ready process should get CPU; also called short-term scheduler

Objectives: a good scheduler should -minimize user response times of all interactive processes (major objective today); maximize system throughput; be fair; avoid starvation

Starvation: happens when some ready processes never get CPU time: typical of schedulers using priorities (lowest-priority processes keep getting set aside). Remedy is to increase the priorities of processes that have waited too long

Fairness: ensuring fairness more difficult than avoiding starvation (give fresh v bad food, no one starves)

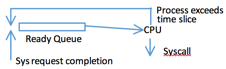
Non-preemptive scheduler: will never remove the CPU from a running process. Will wait until the process releases the CPU b/c: it issues a system call; it terminates. Now obsolete

Examples: First Come First Served (FCFS): simplest & easiest to implement -uses FIFO queue. Seems good but-processes requiring a few ms of CPU time have to wait behind processes that make much bigger demands; Unrealistic. Shortest Job First (SJF): gives CPU to process requesting least amount of CPU time -will reduce avg wait; must know ahead of time how much CPU time each process needs (impossible); still lets processes monopolize CPU

Preemptive schedulers: can temporarily return a running process to the ready queue whenever another process requires that CPU in a more urgent fashion-has been waiting too long; has higher priority. Sole acceptable solution

Preemptive scheduler types: PS w/o priorities: all processes have the same priority. Ready queue is FIFO PS w/priorities: use multiple queues. Differ in the way they adjust process priorities

Round Robin: assumes all processes have same priority -guaranteed to be starvation-free. Similar to FCFS but processes only get the CPU for fixed amount of time *T*CPU -time slice/time quantum. Processes that exceed their time slice return to end of ready queue



Finding right time slice: small time slice means good response time -no process will ever have to wait more than (nready\_queue+1) x *T*CPU time units where nready\_queue is number of processes already in ready queue. Large time slice means better throughput -less context switches Problem: Want to adjust time slice to guarantee maximum waiting time in ready queue *T*CPU = *T*MAX/(nready\_queue+1) -works well as long as sys is lightly loaded; produces small time slices when sys is loaded (too much context switch overhead)

Observation: throughput of sys using RR scheduler actually decreases when its workload exceeds some threshold. Rare among physical systems. Frequent among sys experiencing congestion -Freeway throughput decreases when load exceeds threshold

Solution(I): add priorities. Distinguish among –interactive processes; I/O-bound processes (require small amounts of CPU time); CPU-bound processes (require large amounts of CPU time, i.e. number crunching)

Solution(II): assign –high priorities to interactive processes; med priorities to I/O-bound procs; low priorities to CPU-bound proc

Solution(III): assign –smallest time slices to interactive proc; med TS to I/O-bound proc; biggest TS to CPU-bound proc. Allow higher priority processes to take CPU away from lower priority processes

Outcome: interactive processes will get good response times. CPU-bound processes will get the CPU -less frequently than with RR; for longer periods of time; resulting in less context switch overhead

2 Problems: How to assign priority to processes? Process behaviors may change during their exec. How to avoid starvation? use dynamic priorities. Reward -processes that issue sys calls; processes that interact w/user; processes that have been a long time in ready queue. Penalize -processes that exceed their time slice.

Priority Game: 0 is highest -most UNIX systems, Linux. 0 is Lowest -UNIX sys V.4, Win NT & after.

Implementation: time slice increase when priority decreases -T=high priority; 2T=med priority; 4T=low priority

System V.4 Scheduler: 3 process classes -Real-Time; Time-Sharing; System (for kernel processes). Each process class has its own priority levels; Real-Time processes have highest priority; Time-Sharing has lowest

Real-Time Processes: have fixed priorities -as in Win NT Scheduler. System Admin can define -a different quantum size (rt\_quantum) for each priority level

Time-Sharing Processes(I): have variable priorities. Sys admin can specify the parameters of each priority level -maximum flexibility; maximum risk of making a bad choice. Leaving too many tuning options to sys admin increases chances of sys being out of tune

Time-Sharing Processes(II): parameters include -quantum size (ts\_quantum); new priority for processes that use their whole CPU quantum (ts\_tqexp); new priority for processes returning from waiting state (ts\_slpret).

Time-Sharing Processes(III): maximum amt of time a process can remain in ready queue w/o having its priority recomputed (ts\_maxwait). New priority for processes that have been in the ready queue for ts\_maxwait (ts\_lwait).

System has 4 priority levels: 0 is lowest & 3 is the highest. Or 5 priority levels: 0 is lowest & 4 is highest.

New priorities can be: rewarding a “good” behavior: ts\_slpret & ts\_lwait. Penalizing a CPU “hog”: ts\_tqexp.

How? we increase the priority of processes that -have completed a sys call (might become less CPU bound); have waited a long time in ready queue (to prevent starvation). We decrease priority of processes that -have exhausted their time quantum (they might be more CPU bound)

Guaranteed scheduling: class of scheduling algorithms that want to ensure that its process has its fair share of CPU time. Penalize processes that have already used a large amt of CPU. Most versions of UNIX, Win NT & after, Linux

Old UNIX Scheduler: Priorities take into account past CPU usage.

BSD scheduler: unlike old UNIX scheduler, the BSD scheduler takes into account the system load -old CPU usage is forgiven more slowly when system load is high

Linux Scheduler(I): partitions the CPU time into epochs. At the beginning of each epoch, each process is assigned a time quantum -specifies the maximum CPU time the process can have during that epoch. Processes that exhaust their time slice are not allowed to get CPU time until the next epoch starts.

Linux Scheduler(II): processes that release the CPU before their time quantum is exhausted can get more CPU time during the same epoch. Epoch ends when all ready processes have exhausted their time quanta. Priority of a process is the sum of its base priority plus the amt of CPU time left to the process before its quantum expires.

Windows NT Scheduler: an update of the old VMS scheduler. Scheduler manages threads rather than processes. It has 32 priority levels -16-31 for real-time threads; 0-15 for other threads

Real-Time threads: run at fixed priorities btwn 16-31. Cannot be preempted unless higher priority process arrives in ready queue.

Other Threads(I): run at variable priorities btwn 0-15. Each thread has a base priority -base priority of the thread’s process possibly incremented—or decremented—by a value btwn -2 and 2 (this base priority never changes)

Other Threads(II): thread priorities never go below their base priority. They are -incremented whenever they return from the waiting state; decremented when they exhaust their time slice

More parameters: process attributes include a default processor affinity -specifies default set of processors on which the process’ threads can run. Thread attributes include a thread processor affinity -specifies the set of processors on which thread can run.

Chapter 4: Inter-Process Communication

Types of IPC: message passing (blocking/non-blocking; datagrams, virtual circuits, streams; remote procedure calls); Shared memory

Message Passing: processes that want to exchange data send and receive messages. Any message exchange requires: One send: *send (addr, msg, length);* One receive: *receive (addr, msg, length);*

Advantages: very general -sender and receivers can be on different machines. Relatively secure -receiver can inspect the messages it has received before processing them.

Disadvantages: hard to use: Every data transfer requires a *send()* and a *receive().* Receiving process must expect the *send()* (Might require forking a special thread)

Shared Memory: 2 or more processes share a part of their address space

Advantages: fast and easy to use -the data are there. But -some concurrent accesses to the shared data can result into small disasters; must synchronize access to shared data (topic covered in Ch. 5)

Disadvantages: not a general solution -sender and receivers must be on the same machine. Less secure -processes can directly access a part of the address space of other processes.

Message Passing: defining Issues -Direct/Indirect communication; Blocking/Non-blocking primitives; Exception handling; Quality of Service (unreliable/reliable datagrams; virtual circuits, streams)

Direct Communication: send & receive sys calls always specify processes as destination or source -*send(process, msg, length);, receive(process, msg, &length);.* Most basic solution b/c there is no intermediary btwn sender & receiver. Process executing the receive call must know the identity of all processes likely to send messages -Very bad solution for servers (servers have to answer requests from arbitrary processes). Ex. Phones w/o switchboard (each phone is hardwired to another phone)

Indirect communication(I): send & receive primitives now specify an intermediary entity as destination or source: the mailbox *send(mailbox, msg, size); receive(mailbox, msg, &size);*. Mailbox is a sys obj created by the kernel at the request of a user process. Ex. phones with a switchboard (each phone can receive calls from any other phone). Each phone now has a phone number -callers dial that number, not a person’s name. Taking our phone with us allows us to receive phone calls from everybody.

Indirect communication(II): Different processes can send msgs to the same mailbox -process can receive msgs from processes it doesn’t know anything about; process can wait for msgs coming from different senders (will answer the 1st msg it receives)

Mailboxes: can be –Private (attached to a specific process ex. cell phone); Public (sys objects ex. House plan)

Private mailboxes: process that requested its creations & its children are the only processes that can receive msgs through the mailbox are that process & its children. Cease to exist when the process that requested its creations (& all its children) terminates. Often called ports (ex. BSD sockets)

Public mailboxes: owned by the system. Shared by all the processes having the right to receive msgs through it. Survive the termination of the process that requested their creation. Work best when all processes are on the same machine (Ex. System V UNIX message queues)

Blocking Primitives(I): blocking send does not return until the receiving process has received the message -no buffering is needed; analogous to what is happening when you call somebody who doesn’t have a voice mail

Blocking Primitives(II): blocking receive does not return until a message has been received. Like waiting by the phone for an important msg or staying all day by your mailbox waiting for mail carrier.

Non-Blocking Primitives(I): non-blocking send returns as soon as the msg has been accepted for delivery by the OS -assumes that OS can store the msg in a buffer; like mailing a letter: once the letter is dropped in mailbox, we are done. The mailbox will hold your letter until postal employee picks it up.

Non-Blocking Primitives(II): non-blocking receive returns as soon as it has either retrieved a msg or learned that mailbox is empty -like checking whether your mail has arrived or not

Simulating blocking receives: can simulate a blocking receive w/a non-blocking receive within a loop: *do{ code = receive(mbox, msg, size); sleep(1); //delay} while(code==EMPTY\_BOX);* -known as a busy wait.

Simulating blocking sends: Can simulate a blocking send w/ 2 non-blocking sends & a blocking receive -sender sends message & requests an acknowledgement(ACK); sender waits for ACK from receiver using a blocking receive; receiver sends ACK.

Standard Choice: in general, we prefer; indirect naming; non-blocking sends (sender doesn’t care about what happens once the msg is sent; similar to UNIX delayed writes); blocking receives (receiver needs the data to continue)

Buffering: non-blocking primitives require buffering to let OS store somewhere msgs that have been sent but not yet received. These buffers can have -bounded capacity (refuse to receive msgs when the buffer is full); theoretically unlimited capacity

An Explosive combination(I): blocking receive doesn’t go well w/direct communication -processes can’t wait for msgs from several sources without using special parallel programming constructs (Dijkstra’s alt command)

An Explosive combination (II): using blocking receives w/direct naming doesn’t allow receiving process to receive any msgs from any process but the one it has specified.

Exception condition handling: must specify what to do if one of the tow processes dies -especially important whenever the 2 processes are on 2 diff machines (Must handle: Host failures; Network partitions)

Quality of service: When sender & receiver are on diff machines, msgs -can be lost, corrupted, or duplicated; arrive out of sequence. Can still decide to provide reliable msg delivery

Positive Acknowledgements: basic technique for providing reliable delivery of msgs. Destination process sends an ACK for every msg that was correctly delivered -damaged msgs are ignored. Sender resends any msg that has not been acknowledged within a fixed time frame. 1st scenario – sender: send msg. receiver: send ACK. 2nd scenario – sender: send msg, msg is lost, no ACK is sent, resend msg. 3rd scenario – sender: send msg. receiver: send ACK, ACK is lost. Sender: resend msg. 3rd scenario pt. 2 Receiver must acknowledge a 2nd time the msg -otherwise it would be resent 1 more time. Rule is Acknowledge any msg that doesn’t need to be resent.

Classes of service: Datagrams -msgs are sent one at a time. Virtual circuits -ordered sequence of msgs; connection-oriented service. Streams -ordered sequence of bytes; message boundaries are ignored

Datagrams: Each msg is sent individually -some msgs can be lost, other duplicated or arrive out of sequence; equivalent of a conventional letter. Reliable datagrams -resent until they are acknowledged. Unreliable datagrams

Unreliable datagrams: msgs aren’t acknowledged. Works well when msg requests a reply -reply is implicit ACK of msg. Exactly what we do in real life -we rarely ACK emails & other msgs; we reply to them. Sole reason to ACK a request is when it might take a long time to reply to it.

User Datagram Protocol (UDP): provides an unreliable datagram service -msgs can be lost, duplicated, or arrive out of sequence. Best for short interactions -1 request & 1 reply

Virtual Circuits: establish a logical connection btwn sender & receiver. Msgs are guaranteed to arrive in sequence w/o lost msgs or duplicated msgs -analogous to the words of a phone conversation. Require setting up a virtual connection before sending any data -costlier than datagrams. Best for transmitting large amts of data that require sending several msgs -File transfer protocol(FTP); Hypertext transfer protocol (HTTP)

Streams: like virtual circuits. Do NOT preserve msg boundaries -receiver sees a seamless stream of bytes. Offspring of UNIX philosophy -record boundaries don’t count; msg boundaries shouldn’t count

Transmission Control Protocol (TCP): best known for stream protocol. Provides a reliable stream service. Said to be heavyweight -requires 3 msgs (packets) to establish a virtual connection

Datagrams vs Streams: Datagrams: unreliable, not ordered, lightweight, deliver msgs. Ex. UDP Streams: reliable, ordered, heavyweight, stream-ordered. Ex. TCP

Remote Procedure Calls: apply to client-server model of computation. A typical client server interaction: *send\_req(args); rcv\_req(&args); process(args, &results); send\_reply(results); rcv\_reply(&results);* Very similar to a procedure call to a procedure: *xyz(args, &results); xyz(…){…return; }//xyz.* Try to use the same formalism. We could write *rpc(xyz, args, &results); xyz(…){…return;}//xyz* and let the sys take care of all msg passing details

Advantages: hides all details of msg passing -programmer can focus on the logic of the application. Provides higher level of abstraction. Extends a well-known model of programming -anybody that can use procedures & function can quickly learn to use remote procedure calls. Disadvantage: The illusion is not perfect -RPCs don’t always behave exactly like reg. procedure calls (client & server don’t share the same address space). Programmer must remain aware of these subtle & not so subtle differences.

The user program: contains the user code. Calls the user stub *rpc(xyz, args, &results);* Appears to call the server procedure. The user stub: Procedure generated RPC package -packs arguments into request msg & performs required data conversions (arg marshaling); sends request msg; waits for server’s reply msg; unpacks results & performs required data conversions (arg unmarshaling). The server stub: generic server generated by RPC package -waits for client requests; unpacks request args & performs required data conversions; calls appropriate server procedure; packs results into reply msg & performs required data conversions; Sends reply msg. The server procedure: procedure called by the server stub. Written by the user. Does the actual processing of user requests. Differences w/regular PC: client & server processes don’t share the same address space. Client & server must be on different machines. Must handle partial failures. No shared address space: this means -no global variables; cannot pass addresses (cannot pass arguments by reference; cannot pass dynamic data structures through pointers) Solution: RPC can pass arguments by value & result -pass the current value of the argument to the remote procedure; copy the returned value in the user program. Not the same as passing arguments by reference.

Passing dynamic types: cannot pass dynamic data structures through pointers. Must send a copy of data structure. For a linked list -send array w/elements of linked list plus unpacking instructions.

Architecture considerations: The machine representations of floating point numbers & byte ordering conventions can be different -Little endians start at the least significant byte (Intel’s 80x86 including Pentium); Big-endians start at the most significant byte (Sun’s SPARC and most RISC processors) Solution: Define a network order & convert all numerical variables to that order -use *hton* family of functions; same as requiring all air traffic control communications to be in English Detecting partial failures: the client must detect server failures -can send are you alive? Msgs to the server at fixed time intervals Handling partial executions: client must deal w/the possibility that the server could have crashed after having partially exec the request -ATM calling bank comp (was the account debited or not?).

1st solution: Ignore the problem & always resubmit request that have not been answered -some request may be executed more than once. Will work if all request are idempotent -exec them several times has the same effect as exec them exactly once. Ex. of idempotent requests include -reading n bytes from a fixed location (NOT reading next n bytes); Writing n bytes starting at a fixed location (NOT writing n bytes starting at a current location). Techniques is used by all RPCs in the Sun Microsystems’ Network File System(NFS)

2nd solution: attach each request a serial number -server can detect replays of request it has previously received & refuse to exec them; at most once semantics. Cheap but not perfect -some request could end being partially exec

3rd solution: transaction mechanism -guarantees that each request will either be fully exec or have no effect; all or nothing semantics. Best & costliest solution. Use it in all financial transactions. Ex. Buying a house using mortgage money -can’t get the mortgage w/o having a title to the house; can’t get title w/o paying 1st previous owners; must have mortgage money to pay them. Sale is a complex atomic transaction. Realizations: Sun RPC: dev. by Sun Microsystems; used to implement their NFS. MSRPC (Microsoft RPC) -proprietary version of the DCE/RPC protocol; was used in the distributed component of Obj Model(DCOM). SOAP -exchanges XML-based msgs; runs on the top of HTTP (very portable, very verbose)

Chapter 5: Inter-Process Synchronization

Critical section problem. Mutual exclusion -Software solutions, Hybrid solutions, Semaphores Applications of semaphores. Monitors

Shared memory: 2 or more process share a part of their address space Shared files: 2 or more processes access the same file at the same time The outcome: can expect incorrect results whenever 2 processes (or 2 threads of a process) modify the same data at the same time. 1st ex. Bank account management -each account has a balance; 1 process tries to credit an account; another process tries to debit the same account Problem: have a race condition -outcome of concurrent exec of the processes will depend on process speeds. We will observe irreproducible errors -hard to detect & debug Solution: define critical sections in the 2 functions. Control access to critical sections. 2nd ex. Most txt editors & word processors don’t update directly the file on which they operate -create a temporary *tempfile*; modify the original file when the user issues a save command. What happens if 2 users edit the same file at the same time? Whoever saves first loses their work. Problem: another race condition -should never have 2 users/processes updating the same file at the same time. Don’t let others edit our file -send them a copy

1st problem revisited: bank now keeps accounts in NVRAM in a shared data segment -balances are in a big array indexed by account number

Searching for a solution: 3 approaches: Disabling interrupts: very dangerous solution for user programs. Sometimes used for a very short critical sections in the kernel. Does not work for multithreaded kernels running on multiprocessor architectures. Enforcing mutual exclusion: portions of code where the shared data are read & modified are defined as critical sections. Implement a mechanism that guarantees that 2 processes will never be at the same time in a crucial section for the same shared data. Works best w/short critical regions. Preferred solution. Using atomic transactions: will allow several processes to access the same shared data w/o interfering w/each other. Preferred solution for database access since -most databases are shared -their critical sections can be very long as they often involve a large number of disk accesses

Criteria to Satisfy: Any solution to the critical section problem should satisfy the 4 following criteria: 1.) No 2 processes may be simultaneously into their critical sections for the same shared data (want to enforce mutual exclusion) 2.) No assumptions should be made about the speeds at which the processes will exec.(solution should be general: the actual speeds at which processes complete are often impossible to predict b/c running processes can be interrupted at any time). 3.) No process should be prevented to enter its critical section when no other process is inside its own critical section for the same shared data(should not prevent access to the shared data when it is not necessary). 4.) No process should have to wait forever to enter a critical section(Solution shouldn’t cause starvation). My Mnemonics: solution should provide: 1. Mutual exclusion 2. All the mutual exclusion 3. Nothing but mutual exclusion 4. No starvation

Solutions: 1. Pure software (Peterson’s algorithm) 2. Hybrid (uses *xchg* instruction) 3. Semaphores

Pure Software Solution: Makes no assumption about the hardware. Uses Peterson’s Algorithm. -easier to understand than original solution: Dekker’s algorithm; best way to understand it is going through 3 bad solutions. Peterson Algorithm is a concurrent programming algorithm for mutual exclusion that allows two or more processes to share a single-use resource without conflict, using only shared memory for communication.

#define F 0; #define T 1; //shared var. int reserved[2] = {F,F}; int mustwait;//tiebreaker. Void enter\_region(int pid){ int other; //other process. Other = 1-pid; reserved[pid] =T; //set tiebreaker. Must\_wait=pid; while(reserved[other]&&must\_wait==pid); }//enter\_region void leave\_region(int pid) { reserved[pid] =F; }//leave \_region. Essential part of algorithm is reserved[pid]=T; must\_wait=pid; while(reserved[other]&&must\_wait==pid); When 2 processes arrive in lockstep, last one must wait.

Hybrid Solutions: these solutions use special instructions to achieve an atomic test & set: Putting the lock testing & the lock setting functions into a single instruction makes the 2 steps atomic (cannot be separated by an interrupt) Exchange instruction: assume the existence of a shared lock variable *int lockvar =0; //shared* Will only have 2 possible values -0 meaning nobody is inside the critical section; 1 meaning somebody has entered the critical section. We introduce an atomic instruction *exch register, lockvar* -swaps contents of *register* & *lockvar* How we use it: set register to 1 before doing *exch register*, *lockvar* 2 possible outcomes: If *\*pv==1* -lockvar was already set to 1; we cannot enter the critical section; we must retry. If *\*pv==0* -lockvar was still set to 0; we have successfully set it to 1; we have entered the critical section Before exchange: don’t know state of lockvar. Could be -unlocked & free to grab?; locked & in use by another process? After the exchange: lock was already locked. Attempt failed. Must retry. Lockvar was unlocked. It’s now locked (success). Can enter critical region. Entering a critical region: to enter a critical region, repeat *exchange* until it succeeds *do {exchange(int &plock, int &pv) //plock points to lock var // pv points to register}while (\*pv==1);* Leaving critical section: do *&plock=0;* X86 Xchg instruction: *xchg op1, op2* -exchanges values of 2 operands. Always atomic (implicit *lock* prefix) if one of the operands is a mem addr. -*xchg %eax, lockvar* How to use it? enter\_region: *mov1 1, %eax # set to one. Xchg %eax, lockvar. Test %eax, %eax. jnz enter\_region #try again.* Leave region*: mov1 0, %eax #reset to zero. Xchg %eax, lockvar*. Underlying assumptions: Peterson’s algorithm & all hybrid solutions assume that: -instr. exec. in seq.; Instr. Exec. in an atomic fashion. Less & less true in modern CPU arch. -Intel x86 arch has an instr. prefix lock making any instr. writing into mem atomic. (*lock mov1 1, lockvar*). The bad news: Peterson’s alg. & both hardware sol. We’ve seen rely on busy waits. Busy waits waste CPU cycles -generate unnecessary context switches on single processor arch; slow down progress of other processes. Priority inversion: a high priority process doing a busy wait may prevent a lower priority process to do its work & leave its critical region. Conclusion: must avoid busy waits on single processor architectures. Can use them only for short waits on multiprocessor architectures. Several OS for multiprocessor arch. Offer 2 diff. mutual exclusion mech -busy waits for very short waits; putting the waiting process in the waiting state until the resource becomes free for longer waits.

Semaphores: introduced in 1965 by E. Dijkstra. Semaphores are special integer variables that can be initialized to any value >= 0 & can only be manipulated through 2 atomic operations: P() & V(). Also called wait() & signal() -best to reserve these 2 names for operations on conditions in monitors.

P() Operation: if semaphore value is 0 -wait until value becomes positive. Once value of semaphore is >0 -decrement it. V() Operation: increment the value of the semaphore. How they work: normal implementation is through syscalls: -busy waits are eliminated; processes waiting for a semaphore whose value is 0 are put in the waiting state. Analogy: Paula & Victor work in a restaurant. Paula handles customer arrivals -prevents ppl from entering restaurant when all tables are busy. Victor handles departures -notifies ppl waiting for a table when one becomes available. The semaphore represents the numbers of available tables -initialized w/the total number of tables in restaurant. 2 Problems: Paula can let ppl in when all tables are busy. If ppl leave & don’t notify Victor, table will never be reassigned. Implementation: to avoid busy waits, we’ll implement semaphores as kernel objects. Each semaphore will have a value & an associated queue. New sys calls: *sem\_create()* – creates a semaphore & initializes it, sem\_P() – if semaphore value is > 0, kernel decrements it by 1 & lets calling process continue; otherwise kernel puts calling process in waiting state & stores its pid in semaphore queue. sem\_V() – if semaphore queue is not empty, kernel selects 1 process from queue & puts it in ready queue; otherwise, kernel increments by 1 the semaphore value, sem\_destroy() – destroys a semaphore. Binary semaphores: value can only be 0 or 1. Mostly used to provide mutual exclusion. Semantics of P() operations not affected. V() now sets semaphore value to 1. Mutual Exclusion: assign 1 semaphore to each grp of data that constitutes a critical section. Initial value of semaphore must be 1: *semaphore mutex=1;* Before entering critical region, processes must do -P(&mutex); wait until critical region becomes free. Processes leaving a critical region must do -V(&mutex); Signal the process is leaving the critical section. Making a process wait: initial value of semaphore must be 0. *semaphore waitforme = 0;* Process that needs to wait for another process does: *P(&waitforme);* When the other process is ready, it will do: *V(&waitforme);* Setting up a rendezvous: to force 2 processes to wait for each other, we need 2 semaphores both initialized to 0*. semaphore waitforfirst=0;* *semaphore waitforsecond=0;* When 1st process is ready: *V(&waitforfirst); P(&waitforsecond);* When 2nd process is ready: *V(&waitforsecond); P(&waitforfirst);* Can have deadlock if swapped. Advantages of semaphores: machine independent. Simple, but very general. Work w/any num. of processes. Can have as many critical regions as we want by assigning a different semaphore to each critical region. Can use semaphore for synchronizing processes in an arbitrary fashion. Key idea is layering -pick a powerful & flexible mechanism that apply to many problems; build later better user interfaces. Implementations: UNIX has 2 noteworthy implementations of semaphores -old system V semaphores; newer POSIX semaphores.

Classical Problems: Bounded buffer. Readers & writers. Dining philosophers.

Bounded Buffer: 1 or more producer processes put their output in a bounded buffer -must wait when buffer is full. 1 or more consumer processes take items from the buffer -must wait when buffer is empty. 3 rules: -producers can’t store items in buffer when buffer is full; consumers can’t take items from buffer when the buffer is empty; producers & consumers must access the buffer 1 at a time. Order matters: the order of the 2 P() operations is very important -neither the producer of consumer should request exclusive access to the buffer before being sure they can perform the operation they have to perform. The order of the 2 V() operations doesn’t matter.

Readers-Writer Problem: Have a file (or database) & 2 types of processes: Readers – need to access the file. Writers – need to update it. Readers must be prevented from accessing the file while a writer updates it. Writers must be prevented from accessing the file while any other process accesses it -they require mutual exclusion. Starvation: Solution is favors the readers over the writers -a continuous stream of incoming readers could block writers forever (Result would be writers’ starvation) The Dining philosophers: 5 sit at a table -5 forks causes deadlock. The mutex semaphore prevents deadlock but also prevents 2 non-neighboring philosophers of taking their forks at the same time. The main interest of this problem is its part to the OS folklore.

Limitation of semaphores: they are a low level construct -deadlocks will occur if V() calls are forgotten; mutual exclusion is not guaranteed if P() calls are forgotten. Better solution: need programming lang. construct that guarantees mutual exclusion -will not trust processes accessing the critical region. Can build it on top of semaphores.

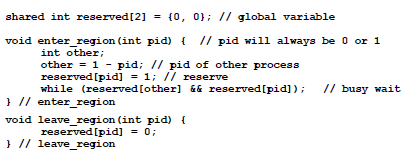
Monitors: programming language construct introduced by Hoare (1974) & Brinch-Hansen (1975). Finally implemented in Java -w/o named conditions. Monitor is a package encapsulating procedures, variables, & data structures. To access the monitor variables, processes must go through 1 of the monitor procedures. Monitor procedures are always exec 1 at a time -mutual exclusion is always guaranteed. Monitor procedures can wait on a condition (cond.wait) until they get a signal (cond.signal) from another monitor procedure. Although conditions look like normal var, the have no value. If a monitor procedure signals a condition and no other procedure is waiting for it, the signal will be lost: It doesn’t help to scream when nobody is listening! If a monitor procedure waits for a condition that has already been signaled, it will remain blocked until the condition is signaled again: It doesn’t help to wait for something that has already happened! Not the same as semaphores: if a process does a V() operation on a semaphore & no other process is doing a P() operation on the semaphore, the value of the semaphore will be changed. This is not the case for condition var. The monitor body: the monitor body is exec. when monitor is started: its major purpose is to initialize the monitor var. and data structures. 1st ex: implementing semaphores on top of monitors -class semaphore w/methods P() & V(). No practical application -monitors are implemented on top of semaphores & not the other way around! Proves that monitors are as powerful as semaphores. Class semaphore{ // private declarations. Private condition not zero; private int value; //semaphore’s value // must be public and syn’d public void synchronized P(){ //check before waiting if (value==0) notzero.wait(); value--; //decrement} //P //must be public & syn’d public void synchronized V() { value++; notzero.signal(); } // V //constructor semaphore(int initial\_val){ value = initial\_val; } //constructor } //class semaphore. 2nd ex: The bounded buffer. Class Bounded\_Buffer w/ methods put() and get(). Class Bounder\_Buffer { //private declarations. Private condition notfull; private condition notempty; private int bufsize; private int nfullslots; //monitor procedures //must be public & sync’d. public void synchronized put(){ //MUST CHECK FIRST if(nfullslots == bufsize) notfull.wait();… nfullslots++; notempty.signal(); } //put //monitor procedures (cont’d) //must be public & sync’d. public void synchronized get(){ // MUST CHECK FIRST if (nfullslots==0) notempty.wait(); nfullslots--; notfull.signal; } //get //constructor is monitor body. Bounded\_Buffer(int size) { nfullslots=0; buffer\_size=size; }// constructor. Semantics of signal: gives immediate control of the monitor to the procedure that was waiting for the signal -the procedure that issued the signal is then put temporarily on hold. Has no effect if there is not procedure waiting for the signal. Causes 2 types of problems. Too many context switches. Prevents programmers from putting signal calls inside their critical sections. Sole truly safe place to put them is at the end of procedure. Not an ideal solution as the programmer can forget to put them there.

The notify primitive: Introduced by Lampson & Redell in Mesa -adopted by Gosling for Java. When a procedure issues a condition.notify(), the procedure that was waiting for the notify doesn’t regain control of the monitor until the procedure that issued the signal –terminates; waits on a condition. Advantages: Fewer context switches. Programmers can put notify() calls anywhere. Very minor disadvantage: Condition might not be true anymore. Should replace if in *if(condition\_is\_false) condition.wait()* By a while *while(condition\_is\_false)condition.wait()*. Java implementation: the java equivalent of a monitor is a Java class whose access methods have been declared synchronized. Java doesn’t support named conditions -when a synchronized method does a wait(), it cannot specify the condition it wants to wait on. Java has notify() and notifyAll().

Old Exams

1. Which of the following statements are *true* or *false* and *why*? a) You cannot combine non-blocking sends with blocking receives. *False*, this is the default for BSD socket calls send() and receive(). b) Peterson’s algorithm for mutual exclusion assumes the existence of a *test-and-set* instruction. *False*, Peterson’s algorithm makes no assumption about the hardware on which it will run. c) Good programmers always put their *notify* operations at the end of their monitor procedures. *False*, notify operations can be put anywhere in your code as they never release the monitor. d) In a RPC package, one of the tasks of the *user stub* is to exchange messages with the user program. *False*, the user stub exchanges messages with the server stub. e) Messages are *reusable resources* since they can be forwarded to other computers*. False*, they are consumed as soon as they have been received. f) Datagrams preserve message boundaries*. True,* messages are sent and received individually.

2. Consider the following solution to the mutual exclusion problem and explain when it fails and what happens then.

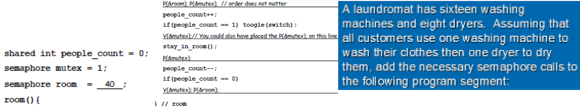
 When two processes try to enter the critical section in lockstep then a deadlock occurs.

3. Consider the function: void doublesquare(int \*pa, int \*pb) { \*pa \*=\*pa; \*pb \*=\*pb; } //doublesquare and assume following calling sequence: alpha = 3; doublesquare (&alpha, &alpha); What will be the value of alpha after the call assuming that:

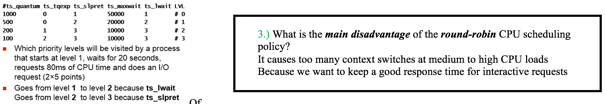
a) the call was a *conventional procedure call*? Alpha = 81 b) the call was a *remote procedure call*? Alpha = 9

4. Give an example of an application where a) Datagrams are more indicated than streams. Short requests from a client to a server b) Streams are more indicated than datagrams. Transferring a large file

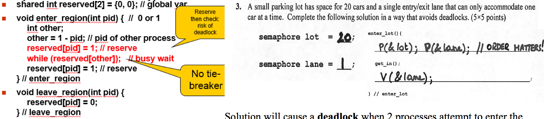
5. What is the major disadvantage of *busy waits*? What can we do to eliminate them? Are there cases where it is better to keep them? a) Busy waits waste CPU cycles and cause unnecessary context switches. b) We should use instead kernel supported solutions that move the waiting processes to the waiting state. c) It is better to keep them in multiprocessor architectures whenever the cost of the wait is less than the cost of the context switches required to bring the waiting process to the waiting state and back. 6. A classroom has two doors and one switch at each door to turn the lights on and off. Complete the following template to ensure that (a) the room will never contain more than 40 people and (b) the light switches will always be on when there are people in the room and off when the room is empty.



1. The problem is a variant of the readers and writers problem discussed in class. Here too, we use a mutex to ensure that people enter or leave the classroom one by one. The big difference is that we do here a P(&room) each time a person enter and a V() each time a person leaves the room in order control the number of people in the room. Semaphore boat =40; semaphore gangway=1; embark(){ P(&boat); P(&gangway); walk(); V(&gangway); }//embark. Debark() { P(&gangway); walk(); V(&gangway); V(&boat); } //debark. 2.



3. Of all *four necessary conditions for deadlocks*, which one is the *easiest to deny*? *Circular wait* 3.) What is the main advantage of datagrams over virtual circuits and streams? Much lower overhead 3.) How can you simulate a blocking receive using only non-blocking primitives? Using a busy wait: while(receive(mbox, buffer, nbytes)==NO\_MSG); 3.) Why do notify primitives are safer to use than the older signal primitives? b/c a notify call never interrupts the procedure calling it. 3.) What is the best way to prevent starvation in scheduling policies implementing variable priorities? We should increase the priorities of processes that have waited for too long in the ready queue. 4. Consider the following solution to the mutual exclusion problem & explain when it fails & what happens then.



Solution will cause a deadlock when 2 processes attempt to enter the critical section in lockstep. 5. consider the function void badexchange(int \*pa, int \*pb) { \*pa=\*pb; \*pb \*= \*pa; } //badexchange. And assume the following calling seq. alpha=3; beta=5; badexchange(&alpha, &beta); conventional procedure call? Alpha=5 & beta=25. A remote procedure call? Alpha=5 & beta=25. 6. What is the main advantage of the all or nothing RPC model over the at most once model? It eliminates the risk of partial executions of the request. Crucial for financial transactions. What is its sole disadvantage? Its high overhead.

1. Consider the following solution to the mutual exclusion problem & explain when it fails and what happens then:

*shared int locked[2] = {0, 0} // global variable void enter\_region(int pid) { // pid will always be 0 or 1 while (locked [1 – pid]); // busy wait locked[pid] = 1; // reserve } // enter\_region void leave\_region (int pid) { locked[pid] = 0; } // leave\_region*

When two processes arrive in lockstep then both processes will enter the critical region

2. Consider the function: *void squarethem (int \*pa, int pb) { \*pa = (\*pa)\*(\*pa); \*pb = (\*pb)\*(\*pb); } // squarethem*

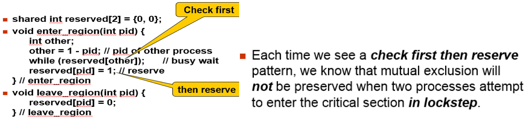
and assume the following sequence: *int alpha = 2; squarethem(&alpha, &alpha);* What will be the value of alpha after the call assuming that the call was: a) A conventional call? Alpha = 16 b) A remote procedure call? Alpha = 4

3. True of False: Most Scheduling policies decrease the priority of processes that have exhausted their slice of CPU time. *True.* Most programmers like to put all their signal operations at the end of their monitor procedures. *True.* Peterson’s algorithm assumes the existence of shared variables. *True.* One cannot initialize binary semaphores. *False.* You cannot combine non-blocking sends and blocking receives. *False.* 4. A cruising boat can carry up to 80 passengers. These passengers can embark or debark through a narrow gangway that can accommodate one person at a time. Complete the two following monitor procedures to ensure that neither the boat nor its gangway will ever be overloaded.

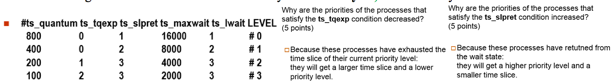
*class Boat { private int passengers; // cannot have more than 80! private condition notfull; public synchronized void embark() {* if npassengers == 80; notfull.wait; npassengers++; walk(); ) // embark *public synchronized void debark() { walk();* npassengers--; notfull.signal; } // debark

6. Questions with short answers: a) What is the *main disadvantage* of the *round-robin* CPU scheduling policy? It causes too may context switches when the system is heavily loaded b) What is the main disadvantage of *spin locks*? They waste CPU cycles while waiting for the lock (and generate context switches.) c) What does it take to prevent deadlocks by denying the *circular wait condition*? We must force all processes to acquire all their resources in the same linear order. d) What is the difference between a *blocking send* and a *non-blocking send*? A blocking sends does not return until the message has been delivered to its recipient. e) How can you implement the *at most once semantics* in a remote procedure call package? We should attach a sequence number to each message sent by a specific client and instruct the server to reject requests with duplicate sequence numbers. f) Why does the *web http protocol* use streams instead of datagrams? Because replies from an http server will not always fit in a single packet and we want these packets to arrive to the client in order without lost packets, damaged packets or duplicates.

1. True of False and why?: a.)The round-robin scheduling algorithm is starvation-free*. True*, Round-robin has no priorities so all processes are treated equally. b.)You can simulate a blocking receive() using a non-blocking receive(). *True*, we can use a busy wait: while(receive()==NO\_MSG); c.)Virtual circuits do not preserve message boundaries. *False*, streams do no preserve message boundaries. d.) Peterson’s algorithm for mutual exclusion assumes the existence of an exchange(XCHG) instruction. *False*, it doesn’t assume the existence of any special instruction. That’s why it’s nontrivial. It only assumes that machine instructions are executed in order and cannot be interrupted. e.) Good programmers always put their signal() operations at the end of their monitor procedures. *True*, Monitor procedures can be interrupted by a waiting process each time they perform a signal operation. f.) Monitor condition are normally initialized to zero. *False*, Monitor conditions have no value and cannot be initialized. 2. Consider the following solution to the mutual exclusion problem & explain when it fails and what happens then.

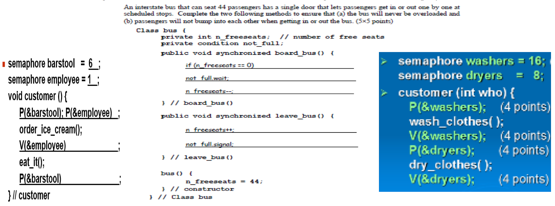


3. What is the major disadvantages of busy waits? Busy waits waste CPU cycles, they generate additional context switches. By what can we replace them? Let the kernel handle process synchronization & put waiting processes in the waiting state. 4. Complete the following sentences: In java, a monitor procedure must start w/ 2 keywords public and synchronized. The main disadvantage of atomic transactions is that they are too costly. 5. Consider a System V release 4 scheduler such as:



Why are the priorities of the processes that satisfy the ts\_lwait condition increased? To prevent starvation.

6. An ice-cream parlor has a single employee selling ice cream and six barstools for its customers. Each employee can only serve one customer at a time and each barstool can only accommodate one customer at a time. Complete the following function template in a way that guarantees that customers will never have to wait for a seat with a melting ice-cream in their hand. Before ordering our ice cream, we must reserve a seat and must get a mutex on the employee.



What is wrong with the following solution to the mutual exclusion problem for two processes?

*int lock[2]; // lock is the only global variable* What happens? Solution will cause a deadlock.

*lock[0] = lock[1] = 0; //0 means FREE* When does it happen? When the 2 processes try to enter the critical

*enter\_region(int pid) { // pid must be 0 or 1* section in lockstep each process will prevent the other from entering.

*lock[pid] = 1; // 1 means BUSY*

*while (lock[1 – pid]);*

*} //enter\_region*

*void leave\_region(int pid) {*

*lock[pid] = 0;*

*} //leave\_region*

Alice and Barbara have to go to New Orleans to present a paper at a computer conference…Their main problem is that they can only communicate through blocking send and receive primitives with direct naming such as: -bsend(Alice, buffer, nbytes) –breceive(Alice, buffer, &nbytes) How will they be able to wait for each other using these two primitives?

Alice will do: bsend(Barbara, buffer, nbytes) Barbara will do: breceive(Alice, buffer, &nbytes)

Complete the following template to obtain a correct solution to the mutual exclusion problem for two processes whose ID’s are either 0 or 1:

*shared int requested[2] = {0, 0};*

*shared int turn;*

*void enter\_region(int mypid) {*

*requested[ mypid ] = 1;*

*turn = 1 – mypid ;*

*while (requested[ 1 – mypid ] && turn != mypid);*

*} // enter\_region*

*void leave\_region(int mypid) {*

*requested[mypid] = 0;*

*} // leave\_region*

Consider the following solution to the mutual exclusion problem:

#define FALSE 0 a) What is wrong with it? It does not guarantee mutual exclusion

#define TRUE 1 b) When does this problem happen? When two processes enter in lockstep

shared int reserved[2] = {FALSE, FALSE};

void enter\_region(int pid) {

while(reserved[1 – pid]); // busy wait

reserved[pid] = TRUE;

} // enter\_region

void leave\_region(int pid\_ {

reserved[pid] = FALSE;

} // leave\_region

1. a) How do you pass a linked list to a remote procedure? You send to the remote server an array containing the elements of the linked list preceded by a header containing the number of elements in the list and how to rebuild it. b) What is the major disadvantage of non-preemptive scheduling policies? Non-preemptive scheduling policies let CPU-bound processes monopolize the CPU. c) What is the difference between blocking sends and non-blocking sends? A blocking send waits until the message is delivered while a non-blocking send does not. d) What is the major disadvantage of the at-most-once semantics for remote procedure calls? The at-most-once semantics for remote procedure calls does not prevent partial executions of calls. e) What is the main advantage of datagrams over streams? They are faster because sending datagrams does not require establishing a connection between the sender and the receiver. f) What is the sole proper way to initialize a mutex semaphore? Set it to ONE.

2. When does the System V Release 4 scheduler: a) Decrease the priority of a process? When a process has exhausted its CPU time slice. b) Increase that priority? When a process returns from the blocked state (“sleep” state). When it has been for too long in the ready queue without getting any CPU time.

3. Consider the function

*void squarethem (int \*pa, int \*pb) {* What will be the value of alpha after the call assuming that the call was:

*\*pa = (\*pa)\*(\*pa);* a) a conventional call? Alpha = 3x3x9 = 81

*\*pb = (\*pb)\*(\*pb);* b) a remote procedure call? Alpha = 3x3 = 9

*} // squarethem*

and assume the following calling sequence:

*int alpha = 3;*

*squarethem (&alpha, &alpha);*

4. Monitor conditions can only have positive values. *False*, they do not have any value. It is generally a good idea to assign fixed priorities to real-time processes. *True*. Most scheduling policies increase the priority of processes that have exhausted their slice of CPU time. *False*, they do just the opposite. It is generally a good idea to assign fixed priorities to real-time processes. *True.* Most scheduling policies increase the priority of processes that have exhausted their slice of CPU time. *False,* they do the opposite. You cannot combine non-blocking sends and blocking receives. *False*, that’s how socket work! V() operations on semaphores are always non-blocking. *True*.

5. Consider a lottery schedule managing four threads. Assuming that thread A has 10 tickets, thread B had 4 tickets and threads C and D have 3 tickets each. a) What are the chances of thread B being elected first among all its peers? 4/(10+4+3+3) = 4/20 = 0.2 or 20 percent b) Show that the scheduler is starvation-free. Every ticket holder will finally get the resource.

6. A diner can seat a maximum of 40 customers. Complete the following code fragments to ensure (a) that the diner will never contain more than 40 customers and (b) that customers arriving together will be seated together.

*class Diner {*

*private int nfree; //diner is full when nfree === 0*

*private condition notfull;*

*public synchronized void arrive (int npatrons) {*

*while (nfree < npatrons)*

*notfull.wait;*

*nfree -= npatrons;*

*} // arrive*

*public synchronized void leave(int npatrons) {*

*nfree += npatrons;*

*notfull.signal;*

*} //leave*

*diner() {*

*nfree = 40;*

*} //constructor*

*} //Diner*