Final Exam

The final exam will focus on material from the second half of the course. There may be questions that relate to concepts in the first part (number systems, addressing, etc) but there will not be any MIPS programming.

The exam will be closed book. No notes, no books, no electronic devices of any kind. The style of the exam will be similar to the midterm and the homework assignments.

The exam will be given in our regular classroom, at 8:30 to 10:20, on Wednesday, June 7.

Review Sheet

The information in this writeup is intended to help you identify the key points that you should be comfortable knowing from the second half of the class (since the midterm). The review sheets are not complete substitutes for the lectures and homework assignments, and so it is possible that there will be questions on the exam that are not completely answered by the material on this review sheet or the midterm review sheet.

Operating Systems

Batch systems run jobs with no on-line user interaction. Multi-programming enables a system to have more than one job in memory at once, and to switch between jobs when one is waiting for I/O to disk or tape. Timesharing adds the ability to switch between jobs rapidly and at short intervals under control of the OS, thereby enabling interactive on-line support. All modern large scale (ie, non-embedded) operating systems implement time-sharing in one form or another.

Real-time operating systems are generally designed to meet "hard real time" requirements in which the system must guarantee a response within a fixed amount of time. Such operating systems generally do very little (if any) dynamic tuning, and the system developer is responsible for directly implementing the performance and priority schemes needed to meet the requirements. A "soft real time" system is one that can run some threads at a high priority so that they get to run quickly without much interruption, but there are no absolute guarantees about performance in such a system.

Parallelism is supported in many forms. Tightly coupled systems generally have multiple CPUs on one system board (2, 4, 8 processors are typical) which share memory and can communicate quickly. A single OS schedules threads on all the CPUs in the system. Loosely coupled systems generally have a network connecting the system nodes. The network can be small and high speed (generally referred to as a clustered system) or it can be larger and slower (generally referred to as a distributed system). The OS running on each node schedules threads on that node, perhaps with input from the other nodes.

Processes and Threads

A process is an active instance of a program. There may be more than one process running the same program at the same time. A process defines an address space and various parameters that the operating system needs such as the associated user, open files, current working directory, etc. The word "process" is often used in a general sense to mean a program running on a computer. On systems that support multi-threading, the word process is used to refer to the address space and associated parameters, but not to the information required for scheduling and executing (such as the priority, program counter, registers, and stack pointer). On a multi-threaded system, those data items are kept on a thread-by-thread basis, and there may be more than one thread per process. On a system that does not support multi-threading, scheduling information is kept for each process.

Creating a new process is considered to be a heavyweight operation. The address space must be allocated and registered with the operating system, and all the other control structures must be allocated and initialized. Usually, a program file must be read in from disk and stored in memory when a new process is created.

Threads are the objects that are actually scheduled on a multi-threading system. Associated with each thread is the information needed to implement a "thread of control." This includes most particularly a program counter value, a stack pointer value, and values for all the CPU registers. The operating system also maintains information associated with the thread scheduling such as priority, current scheduling state, and various control parameters.

Threads are considered to be lightweight objects. Some of the information that the system needs about a thread is common to all threads in a process, and so that information is kept at the process level and does not need to be allocated and initialized when a thread is created. Also, because threads in one process share a common address space, they can communicate efficiently using memory without having to go through the operating system with its associated overhead.

Each of the threads in a process has its own stack. This means that there are several sections of memory that are dynamically growing and shrinking, but as long as the area allocated for each stack is large relative to the amount of space required on the stack during execution it is not a problem.

The main benefit of threads is that they provide a relatively easy way to provide concurrency while not incurring the higher overhead costs associated with parallel processes. This concurrency can be used to enhance the responsiveness of interactive programs, facilitate scheduling on multiple CPUs, and provide parallelism to server applications that perform a small set of functions for many connected users in parallel.

Threads can be defined and scheduled entirely in the user program (user threads) or they can be defined and scheduled in the operating system (kernel threads). The advantage of user threads is that no operating system involvement is needed when switching threads, and so scheduling and switching threads can be fairly fast. The disadvantage is that the operating system doesn't know that there are multiple threads, and so it is scheduling the whole process as one unit. It may

make bad decisions such as blocking the whole process because one thread is blocked for I/O, or running a process even though the only thread that is able to run is an idle thread. The advantage of kernel threads is that the OS can schedule threads individually, regardless of what other threads in the process are doing. This allows better scheduling decisions, and also permits threads from one process to run concurrently on different CPUs in a multi-CPU system. The disadvantage of kernel threads is that a trip through the operating system is necessary in order to perform thread scheduling.

Scheduling

Scheduling is the mechanism by which the operating system decides which task to run next and for how long.

Switching between tasks is a context switch. All information needed in order for the new task to run must change on every context switch, and so this can be a fairly expensive operation.

On a multi-threaded system with kernel threads the scheduler selects the next thread to run. On a system with just user threads or no threading at all, the scheduler selects the next process to run. The word "task" is often used as a synonym for "the object being scheduled" when the distinction between thread and process is not relevant.

The common thread states are ready (can run if there is a processor available), running (currently has control of a processor), and waiting (waiting for some event such as I/O completion). An OS often defines additional states to supplement these for implementation convenience. The threads that are in a particular state are kept on queues. The operating system links the thread information into a queue to record the current state, and then re-links it to another queue to record a state change. The thread to run is selected from one of the ready queues, according to the scheduling algorithm.

The goals of the scheduling algorithm are to maximize throughput and resource utilization, and to provide good response time, wait time, and turnaround time. These goals are often in conflict, and so scheduling is a usually a dynamic process that adjusts to the workload according to various system tuning settings.

Preemptive scheduling is implemented by allowing the OS to gain control of the system on a regular basis using a clock interrupt. The scheduler selects the next thread to run, and if it needs to, it can evict the thread that currently has control of the CPU. This approach enables the system to control overall system performance.

Non-preemptive scheduling relies on the running thread releasing control of the CPU voluntarily, either by making an I/O call or by explicitly yielding. This reduces the amount of control the system has, but is appropriate in some real time or dedicated applications where the program design explicitly incorporates knowledge about scheduling requirements.

CPU-bound tasks tend to use the CPU for long periods with no break for device I/O. An I/O-bound task executes relatively few instructions between I/O calls, and can often complete a

processing burst and start another I/O call with just a quick turn as the running thread. CPU-bound tasks are generally scheduled at a lower priority than I/O-bound tasks so that system responsiveness is kept relatively high, while maintaining reasonable throughput. The I/O referred to here can be to any external device such as a disk, a tape drive, a keyboard or other user input device, or a network interface.

Most modern scheduling algorithms are a variation on multi-level feedback queues. There are multiple priority levels, which have associated ready queues. The scheduler monitors threads and modifies priorities based on thread behavior, thereby changing the order and frequency with which threads get access to the CPU. Priority boosting and decay are terms describing the changes that the scheduler makes to thread priorities.

Quantum is a word used to describe the length of time a thread is granted access to the CPU when it is allowed to run. A scheduler sometimes increases the quantum to allow a thread more time on the CPU once it is selected to run so that it is more likely to finish the work it has to do before it gets switched out again.

Synchronization

When multiple threads of execution share a block of state information, it is critical that updates to the shared state be handled sequentially, and not simultaneously. Since a thread can be interrupted at any time by the operating system, there are explicit mechanisms by which threads can coordinate their access to shared state.

A lock provides mutual exclusion. A thread must acquire the lock before using the shared state information, and then release the lock when it is done. In this way, it is certain that only one thread will have access at a time, because the acquire function will wait for the lock to be released if necessary.

An atomic read-modify-write capability is required in order to implement locks. This capability is built into the hardware at the instruction level, and guarantees that a thread can try to acquire a lock and then correctly determine whether or not it succeeded, no matter when a context switch might happen.

Busy-waiting describes having a thread sit in a tight loop, waiting for a lock to be released. This is simple to do, but very inefficient in most cases because no useful work is being done while the loop is executing.

High level constructs that support coordinated access to shared state include locks (critical section, mutex) and monitors (condition variables). The basic lock provides mutual exclusion, but does not provide a good (efficient) way of waiting for changes in the shared state.

Monitors (condition variables) add the ability to wait for changes in state. In order to use a condition variable, a thread acquires the lock that protects the shared state, and checks the values of interest. If the required condition is not true, then the thread calls the wait function. The wait function releases the lock and adds the thread to a wait queue for this condition. If the required

condition is true, then the thread can perform its task and then release the lock. If the thread makes any changes to the shared state, it uses the signal or broadcast function to indicate that one or all waiting threads should be allowed to run and check the condition again. One or all of the waiting threads are made ready, and eventually run again. When a waiting thread returns from the wait call, it again has possession of the associated lock, and can check for the required condition. The condition may or may not be true, so it is important that the check and wait be made in a while loop, rather than a single if statement.

You should be familiar with the use of locks and condition variables as illustrated in the synchronization lectures dated May 17 and May 19, and be able to answer straightforward questions based on similar examples.

Deadlocks

In any situation where there are tasks and resources that the tasks need to accomplish their work, there is the potential for deadlock. The necessary conditions for deadlock are mutual exclusion, hold and wait, no preemption, and circular wait. You should be familiar with these conditions and be able to discuss solutions for them, given a description of a deadlocked situation.

Memory Management and Virtual Memory

A program address (or virtual address) is the address that a program uses internally to reference a location. A physical address refers to a specific physical location in memory. On small systems, the two addresses may be the same, but in most systems they are not.

The ability to decouple the virtual address space as perceived by a running program from the actual physical memory available is a key benefit of the virtual memory scheme.

Paged allocation and base and bounds allocation are two schemes for allocating memory to processes. Be familiar with the paging diagrams in the memory management lectures of May 24 and May 26, and be able to answer questions about memory allocation and page tables based on similar diagrams. Be familiar with the use of flat and multi-level page tables to translate from a virtual page number to a physical page number.

Bringing a page into memory from disk when it is referenced is called demand paging. A page fault occurs, the system determines that the program address is valid but the page is not in memory, and the OS reads the page in and then restarts the program at the instruction that caused the fault. FIFO, RANDOM, OPT (or MIN), LRU, and CLOCK are all algorithms by which the OS can decide which page should be evicted. OPT chooses the page which will not be used for the longest time. Since this is not something that we can accurately determine, the LRU and CLOCK algorithms are approximations to it.

Thrashing occurs when pages are removed from memory but are needed again very quickly. The constant eviction and rereading causes a significant slowdown.

It is common for a program to be actively using much less memory than it potentially might use. This means that a small working set of pages in memory is often enough to let a program run efficiently even though the program address space that the program defines might be much larger. An example of the difference between active use and potential use is the allocation of a large virtual address range for a stack, and the small amount of physical memory that is actually needed during execution for most stacks. Generally, stacks are allocated a small amount of physical memory initially, and then expand using page faults and demand paging.

External fragmentation (unused space is not available for allocation because it is broken up in small pieces) and internal fragmentation (unused space is wasted by being included in the basic allocation block size) are important considerations in memory management efficiency.

File Systems

Files are an abstraction that allows us to define and manipulate a collection of bytes on a storage device. Each file is listed in a directory. Files are stored on disk in blocks. Blocks are the basic allocation unit on a disk, and they are one or more sectors in length. More sectors per block means that more data can be read per access, and allows larger file sizes when using fixed size index tables to allocate blocks to files. Increasing the block size also increases the internal fragmentation.

Contiguous allocation means that all the blocks in a file are allocated sequentially in one track or in several adjacent tracks. This scheme generally provides fast access, but limits how files can be located and expanded.

Linked allocation is more flexible in how files are placed on the disk, but requires more seeking in order to read through the file, is extremely inefficient for random access, and requires some overhead in each block.

Indexed allocation uses an array of pointers to show which blocks are allocated to a file. This is generally efficient, although there are issues associated with how big the index can grow and where the index is actually located on the disk relative to the file.

Operating systems and disk drives use a variety of caching and request scheduling techniques to reduce the amount of time that the system must wait for the mechanical time delays (seek time, rotational delay, media transfer time) involved in disk I/O. Common request scheduling algorithms include Shortest Seek Time First (potential starvation issue) and SCAN / Circular-SCAN.